DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, *SWEDEN 2019*

Sort Merge Buckets: Optimizing Repeated Skewed Joins in Dataflow

ANDREA NARDELLI

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Sort Merge Buckets: Optimizing Repeated Skewed Joins in Dataflow

ANDREA NARDELLI

Master in Computer Science
Date: 21st June 2019
Supervisor: Amir H. Payberah
Examiner: Vladimir Vlassov
School of Electrical Engineering and Computer Science
Host company: Spotify AB

Abstract

The amount of data being generated and consumed by today's systems and applications is staggering and increasing at a vertiginous rate. Many businesses and entities rely on the analysis and the insights gained from this data to deliver their service. Due to the massive scale of this data, it is not possible to process it on a single machine, requiring instead parallel processing on multiple workers through horizontal scaling. However, even simple operations become complicated in a parallel environment. One such operation are joins, used widely in order to connect data by matching on the value of a shared key. Data-intensive platforms are used in order to make it easier to perform this and other operations at scale. In 2004, MapReduce was presented, revolutionizing the field by introducing a simpler programming model and a fault-tolerant and scalable execution framework. MapReduce's legacy went on to inspire many processing frameworks, including contemporary ones such as Dataflow, used in this work. The Dataflow programming model (2015) is a unified programming model for parallel processing of data-at-rest and data-in-motion. Despite much work going into optimizing joins in parallel processing, few tackle the problem from a data perspective rather than an engine perspective, tying solutions to the execution engine. The reference implementation of Dataflow, Apache Beam, abstracts the execution engine away, requiring solutions that are platform-independent. This work addresses the optimization of repeated joins, in which the same operation is repeated multiple times by different consumers, e.g., user-specific decryption. These joins might also be skewed, creating uneven work distribution among the workers with a negative impact on performance. The solution introduced, sort merge buckets, is tested on Cloud Dataflow, the platform that implements the eponymous model, achieving promising results compared to the baseline both in terms of compute resources and network traffic. Sort merge buckets uses fewer CPU resources after two join operations and shuffles fewer data after four, for non-skewed inputs. Skew-adjusted sort merge buckets is robust to all types and degrees of skewness tested, and is better than a single join operation in cases of extreme skew.

Sammanfattning

Mängden data som genereras av applikationer och system ökar med en acceleration som inte tidigare skådats. Trots mängden data måste företag och organisationer kunna dra rätt slutsater av sin data, även om mängden är så stor att det går att behandla på en dator. Istället behövs parallella system för att bearbeta data, men de enklaste operationerna blir lätt komplicerade i ett parallellt system. En sådan enkel operation är join, som grupperar matchande par av datarader för en gemensam nyckel. Processningsramverk har implementerat join och andra operationer för att underlätta utveckling av storskaliga parallella system. MapReduce, som är ett sådant ramverk, presenterades 2004 och var banbrytande genom att tillhandahålla en enkel modell för programmering och en robust och skalbar exekveringsmiljö. MapReduce lade grunden för fler ramverk, till exempel Dataflow som används i denna uppsats. Dataflow (2015) är en programmeringsmodell för att parallellt behandla lagrad data på hårddisk och strömmande data. Join är en kostsam operation och trots att mycket arbete läggs på att optimera join i parallell databehandling, angriper få problemet från ett dataperspektiv istället för att optimera exekveringskod. Apache Beam, referensimplementationen av Dataflow, abstraherar bort exekveringsmiljön och ger utvecklare möjligheten att skriva databehandlingskod som är oberoende av platformen där den exekveras. Denna uppsats utforskar metoder för att optimera joins som utförs på ett repeterande sätt, där operationen utförs på en datamängd, men flera gånger av olika data-pipelines. Ett exempel på en sådan operation är kryptering av användarspecifik data. Join utförs ibland på data som är skev, det vill säga där vissa join-nycklar förekommer oftare än andra, vilket ofta leder till en negativ effekt på prestanda. Sort Merge Bucket Join, en optimering av join operationen och en lösning för skeva datamängder, introduceras i denna uppsats med tillhörande implementation för Cloud Dataflow. Resultaten av denna optimering är lovande med anseende till minskad användning av resurser för processning och nätverkstrafik.

Acknowledgments

Thanks a lot to all the people at Spotify, in particular Flávio Santos and Viktor Jonsson, for their continuous support during this thesis work.

My great appreciation goes to Amir Payberah for his guidance and assistance all through this project.

Finally, I would like to extend my gratitude to all my friends, Susanna, and my mom for putting up with me in these sometimes challenging months.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In recent years, the words *big data* have been used to refer to datasets that are too large to process through conventional approaches. As many businesses rely on analyzing this data in order to e.g., obtain insights about their processes and have a better understanding of their users, research in the field of data-intensive computing is of high interest and utility.

Spotify is the world's most popular music streaming service with over 200 million active users [1]. Every time a user performs an action (e.g. searching a song, creating a playlist, ...), a small piece of information, an *event*, is generated and collected by Spotify's systems. The scale of collected data is massive: billions of events per day (average of millions every second). These datasets are too large to be processed on a single machine, requiring instead processing on *data-intensive platforms* consisting of multiple processing units working in parallel. Events contain specific information such as which song finished playing, which user was playing that song, which client (Android, web browser, ...) that user was using, and more. As part of the event delivery infrastructure, the data undergoes transformations such as deduplication and encryption by *joining* with other massive datasets. The programs responsible for these tasks are written in the *Dataflow programming model* [2], and are called *pipelines*.

Joins are a relational operation, in which the records of two tables are matched based on the value of a shared join key. As an example, an `artist` dataset may be joined with the `song` dataset over a shared join key, the `artist_id`, to compute artist collaborations. Joins are a standard operation for relational databases, but present several scalability challenges when the magnitude of data is massive [3]. In a distributed, parallel share-nothing architecture these challenges include network communication, as workers hold a certain fraction of the records and must communicate with each other in order

to produce a join with correct results. As a consequence, compared to traditional processing in-memory on a single machine, these operations also have a much higher latency. Other challenges such as uneven data distribution among the workers, known as *skewness* [4], and failures (such as hardware failures) may result in *stragglers*. Stragglers are processing units that are struggling to process data and delaying pipeline completion.

In order to maintain users' privacy and comply with privacy regulations, all of Spotify's sensitive data is stored in encrypted format using a set of encryption keys unique to each user [5]. As an example, the data is stored in the `events` table, while user keys are stored in the `user_keys` table. Each event consists of a `user_id` and a variety of fields dependant on the event type, while each entry in the `user_keys` table consists of the `user_id` and a `user_key`. Batches of incoming events are joined with the `user_keys` table on the `user_id` in order to be encrypted with the corresponding `user_key` before storage. Many of Spotify's features (such as user-specific recommendations) are powered by processing the events, which however must first be decrypted by joining again with the user encryption keys. As it is not possible to store unencrypted data and storing the keys with the events would defeat the purpose, all downstream consumers must first join the two tables and perform the decryption operation before processing. The en/decryption operations consist of joins that, as briefly explained before, are expensive in terms of time and usually dominate the execution time of the pipelines. Over the years, the number of pipelines that depend on decrypting data is increasing, as data is being mined for new insights. Hence, optimizations of these operations can present improvements in performance or reduction of cost for Spotify and more broadly for all that are engaging with distributed processing.

1.1 Research Question

This work addresses two research questions:

1. How to optimize *repeated* joining operations in the Dataflow model? Repeated describes the scenario as presented above, in which e.g. data can not be stored unencrypted and must be decrypted every time before processing via joining. Optimization refers to improving performance w.r.t. multiple measurable quantities such as wall time, CPU time, network and storage costs.

2. How to handle skewness in joins in Dataflow pipelines? For example, a skewed dataset may contain a join key that is particularly frequent and

might overload a worker, creating stragglers. Note that skewness and its consequences can arise even if joins are not repeated.

1.2 Goals

Correspondingly with the research question, the goals of this work are twofold:

1. First, it aims to optimize repeated join operations in a distributed setting. This optimization is called **sort-merge buckets** (SMB) with the associated sort-merge buckets join operation.

2. Secondly, it tackles the issue of skewness in Dataflow applications and proposes a solution in the SMB framework.

1.3 Method

This work adopts an empirical methodology, due to the high complexity and number of interconnected systems which need to be taken into account when evaluating data-intensive platforms. At first, we introduce the sort-merge bucket solution. Then, the solution is extended to handle skewed data and address some limitations.

Sort-Merge Buckets (SMB)

In order to optimize the repeated join use-case, the main idea consists of preprocessing the data so that it is easier to join later. The preprocessing consists of sorting the data such that a merge join algorithm may be used afterwards, instead of a hash join. From the perspective of network IO, a merge join does not require any *shuffling* as it simply scans both sides of the join in an interleaved fashion while outputting groups that satisfy the join condition, whereas a hash join would need to hash the join key of each record in order to determine which record is responsible for that key. Shuffling is explained in Chapter 2: it refers to the process of forwarding records to the correct worker. As the communication step has been removed at joining time when using a merge join, it is expected that the cost of sorting the data is amortized in a fixed number of repeated joins. However, the scale of data introduces several problems with regards to sorting the datasets, as will be elaborated on Chapter 3.

In order to overcome these problems, the data is instead partitioned in smaller *buckets*, each of which is sorted. Much like a hash join would, the

corresponding bucket for a record is determined through a hash function on the join key. Instead of simply sorting the data, the preprocessing step (also referred to as *bucketing* step) consists of creating these sorted buckets.

The dataset on the other side of the join is then preprocessed in the same way for the same number of buckets and the same hash function. At this point, the *joining* step simply becomes joining the respective buckets from the two datasets through a merge join, as each bucket is sorted. However, this technique still suffers skewness and has added complexity in the form of the parameter for the number of buckets. In order to overcome these problems and tackle the second research question, the sort-merge buckets solution is extended.

Skew-Adjusted SMB

A common approach in handling skew in Dataflow or similar parallel processing systems consists of replicating data. Consider for example a join between a skewed and a non-skewed dataset. The skewed data can be distributed randomly across all workers, each of which receives a copy of the other side of the join. If the source of skewness is known ahead of time, it is not necessary to replicate the entirety of the non-skewed dataset. However, when both of these datasets are very large and no assumptions can be made on types and degrees of skew, this approach will not work as the amount of replicated data is linear in the number of workers.

A similar idea of replicating data can be applied to SMB with a few modifications. First, instead of setting a fixed number of buckets, a target size for each bucket is set. A bucket is skewed if its contents would be greater than the target bucket size. In this situation, the bucket is divided into a number of *shards*, each of which is sorted. When joining two buckets, buckets that are sharded compute the merge join over all pairs of shards.

In this scenario, the amount of replicated data is linear in the number of shards on the other side of the join operation. Depending on number of workers and size of each bucket this approach can be preferable to the regular replication approach as the main “knob” for scalability in distributed processing consists of more workers (horizontal scaling) as opposed to bigger workers (vertical scaling): replication of data in the skew-adjusted SMB depends only on skewness of the other side of the join.

1.4 Results

The solution is tested on two different types of datasets. First, we evaluate it on real data from Spotify's event delivery infrastructure. Sort merge buckets uses less compute resources after two joins, and shuffles fewer data after four. This dataset, however, does not suffer from skewness under normal operating conditions. Hence, in order to perform a thorough evaluation, we generate a similar dataset with increasing degrees of skew. While the performance of regular join operations and sort-merge bucket degrades quickly with medium degree of skew, skew-adjusted SMB proves to be effective even when high degree of skew is present. In such scenarios of high skew, skew-adjusted SMB performs better than regular joins even for a single join operation.

The solution is backwards-compatible with existing datasets, as the pre-processing step of SMB simply consists of reordering the data by writing it out in sorted buckets. A user unaware of the underlying structure can still join the data using standard operations. As a result, SMB is also not tied to the execution platform and applicable in other data-intensive platforms. Lastly, skew-adjusted SMB is scalable and robust to all types and degrees of skew tested in this project. It does so by engineering the data in such a way to prevent the phenomenon of stragglers.

As a result of just preprocessing the data by sorting, a considerable improvement in compression ratio is also achieved, reducing the size of a real-world 1.5TB dataset to a 1.1TB dataset.

1.5 Benefits, Ethics and Sustainability

In the wake of Europe's General Data Protection Regulation (GDPR) on 25th May 2018, privacy and sensitive user data has become a broadly discussed topic. The text of GDPR mentions how "appropriate technical and organisational measures" must be put in place in order to safeguard sensitive data through a series of principles such as data protection, privacy-by-default and informed consent. GDPR does not however specify the implementation details of these principles. In Spotify, the scenario described in this thesis work (repeated joins) arises when dealing with en/decryption of sensitive data. It is believable that other businesses have adopted similar techniques in their implementation of GDPR. This work can make these implementations faster and have better performance, enabling a wide and efficient compliance with the regulation.

It is imperative that the optimization presented in this work do not come

at a price of correctness of the operation. When evaluating this methodology, care must be given in order to ensure that results are correct and e.g., privacy is not compromised.

The vertiginous growth in the amount of data is impacting the energy industry. Many of the world's businesses are powered by data-intensive processes in data centers, amounting to almost 3% of the world's total electricity [6]. Due to the widespread need to match data through joins, this work contributes to creating more efficient and sustainable data processing at scale.

1.6 Outline

Chapter 2 describes the theoretical framework of this project while analysing related work in this field. Chapter 3 analyses the design and implementation of relevant data-intensive platforms, including the one used for this project. Chapter 4 presents the methodology used in this project and solution used in this degree project. Chapter 5 describes the results of the work. Chapter 6 discusses and evaluates this work while presenting possible future work.

Chapter 2

Background

The growth of data over recent years has been staggering, with forecasts predicting the total amount of data to increase from 33 zettabytes in 2018 to 175 zettabytes by 2025 [7] (one zettabyte is equivalent to a trillion gigabytes). This "big data" is characterized by four dimensions:

1. **Volume**: the amount of data.
2. **Velocity**: the rate at which data is being generated.
3. **Variety**: the heterogeneity of data.
4. **Veracity**: the uncertainty of data, e.g., due to its quality.

The field of data-intensive computing studies how to store and process data at this scale, whereas the systems which handle the data are known as data-intensive platforms. As the scale is too large to process on a single machine, these platforms consist of multiple units working in parallel.

The remainder of this chapter is organized as follows. Section 2.1 presents a brief history of the systems that led to the Dataflow model, used in this work. Section 2.2 presents join algorithms and their parallel counterparts in Dataflow. The concept of SMB reuses several ideas from these algorithms. Section 2.3 presents the concept of skewness. Section 2.4 presents related work.

2.1 Dataflow Programming

Before delving into the Dataflow programming model [2], it is important to briefly go over its predecessors and the main contributions that have led to it. The programming model of a data-intensive platform describes the operations

available to users of that system (i.e. the programming "style"). The models and the underlying platforms have been created with the same motivation: processing of data that is at massive scale. In these scenarios, scaling up is impossible or prohibitively expensive and the data is such that it can not fit in a single machine, hence the need to store and process it across multiple machines organized in clusters. Distributed computation and the design of data-intensive platforms at this scale is a hard and interesting problem presenting several challenges:

1. **Performance**: the efficiency in executing a task, for example in terms of time, cost, or network communication.

2. **Fault tolerance**: the robustness of the platform in presence of software or hardware faults and its ability to recover from these faults.

3. **Scalability**: the ability to scale the platform by adding or removing workers.

Contributions to these challenges are important: users can benefit both from a more transparent processing model and a more efficient platform.

One of the most significant contributions in the field of data-intensive computing is MapReduce [8]. In 2004, the MapReduce programming model revolutionized share-nothing parallel processing due to its simplicity and scalability. Soon after the publication, Apache Hadoop, an open-source implementation of MapReduce was created (its first version, $0.1.0$, was released in April 2006) and is still developed to this day. The original authors of [8] present MapReduce and its underlying implementation as a platform for parallel, scalable, fault tolerant data processing. A MapReduce program, or job, can be broken up in a series of stages in which the user must specify the two eponymous operations (*map* and *reduce*). In detail, the stages are:

1. The read stage – read input data.
   For example, read each line of a text file and produce (key, value) pairs (k_1, v_1) where k_1 is the offset inside the file, and v_1 is the line.

2. The map stage – $\texttt{map} : (k_1, v_1) \longrightarrow \texttt{List}(k_2, v_2)$
   The user must implement this operation, which receives pairs (k_1, v_1) as input and produces a $\texttt{List}(k_2, v_2)$ as output. Each input pair is hence mapped into a list of 0 or more output pairs of possibly different types.

3. The shuffle stage – connects the map stage to the reduce stage.
   This communication step is responsible for forwarding records with the

same key k_2 to the same worker. In Hadoop, this is achieved through a hash function on k_2 modulo the number of available workers for the reduce stage (also called reducers). This step is expensive as it requires network communication, which is slower than direct memory access.

4. The reduce stage – `reduce` : $(k_2, \texttt{List}(v_2)) \longrightarrow \texttt{List}(v_2)$
   The user must implement this operation, which receives pairs $(k_2, \texttt{List}(v_2))$ as input and produces a $\texttt{List}(v_2)$ as output.

5. The output stage – writes the $\texttt{List}(v_2)$ in the desired output format.

Whereas MapReduce manages the need for distributed processing, it is important to mention how it is deeply connected to the underlying filesystem it was developed with, which handles the need for distributed storage. Much like MapReduce inspired Hadoop, the Google File System [9] inspired its open-source counterpart, the Hadoop Distributed File System [10]. Many of the design decisions from the first versions of Hadoop are interconnected with HDFS. For example, earlier versions of Apache Hadoop used replication, as provided by HDFS, in order to have fault tolerance.

The stages in a MapReduce job are broken into partial *tasks*, which are scheduled to the workers in the cluster. A task can be thought of as a work "unit", for which a single worker is responsible for. How tasks are split and how they are scheduled to the workers is implementation-dependant, but in general the goal is creating relatively small tasks have similar load in order to easily distribute them across the cluster. As an example, early versions of Hadoop used a `FIFO` scheduler for tasks, each block on HDFS (equivalent to 64 megabytes) was mapped to a `map` task and a `reduce` task was created for each unique k_2. The sizes of these tasks have a direct correlation with skewness: if some task is too large, it will create a straggler that slows down completion for the whole stage.

The initial version of Hadoop presented several drawbacks. Compared to the systems of today, Hadoop had relatively long processing times and latency. Surely performance improved as newer versions were released, but part of it were due to design choices e.g., the intermediate output from each stage was persisted on the local storage of each worker machine, and was later fetched as part of the following stage which is a slow network-dependant procedure. Furthermore, the programming model was fairly limited, with operations only being able to be expressed in terms of `map` or `reduce`, exactly once and exactly in this order (for example, multiple `map` operations required multiple MapReduce jobs). Other disadvantages, such as the single point of

failure of the master in the master-slave cluster configuration or the overhead of replication for fault tolerance were amended by recent versions of Hadoop and HDFS.

Over the years, MapReduce became the "golden standard" of parallel processing systems and its legacy is still strong today by inspiring all works that came after it. Figure 2.1 shows a timeline with some of the major advancements described in this section starting from MapReduce. Some of these consist of building on top of MapReduce, while others try to overcome shortcomings that were inherent in the design of MapReduce. Note that many other contributions and platforms exists but are omitted in this timeline for brevity. Entries on the left of the left represent contributions or products by Google, whereas entries on the right represent open-source projects from the Apache Software Foundation. For a comparison between Cloud Dataflow and other data-intensive platforms, see Section 3.2. Two key contributions to the Dataflow model, also initially developed at Google, are:

- **FlumeJava** [11]: FlumeJava is a Java library created to make it easier to write parallel processing jobs. Its main contribution is the introduction and use of `PCollections`, immutable distributed data structures which support a variety of operations. These data structures contain records which are automatically distributed among all worker nodes, removing the burden on the programmer of writing parallel code. The operations on `PCollections`, called `PTransform`, are not executed immediately, defining instead an execution plan which is optimized before being executed. The execution plan can be represented as a directed acyclic graph (DAG) in which data flows from `PTransform` to `PTransform`. The DAG abstraction became very popular, and is used in all data-intensive platforms today. When used on top of MapReduce, the operations called on the `PCollections` are translated into a series of MapReduce jobs. Optimizing the execution plan allows to fuse some operations together in order to reduce the number of underlying `map` or `reduce` steps. The programming interface for FlumeJava is for the most part responsible for *Apache Beam* [12], the reference API which implements the Dataflow model. FlumeJava's open-source counterpart is Apache Crunch, but its notion of `PCollection` went on to inspire the concept of Resilient Distributed Datasets (RDDs) [13], which are at the core of Apache Spark. Apache Spark is a data-intensive platform, originally released in 2012, that improves on MapReduce and is widely used today. For a comparison with Dataflow and other platforms, see Section 3.2.

- **MillWheel** [14]: The systems discussed so far have been described in the context of processing data-at-rest (batch processing), as opposed to data-in-motion (stream processing). Attempts to extend the batch-processing model of Hadoop and Spark resulted in systems like Spark Streaming [15], with limited flexibility and limitations on predefined operators and user-defined applications. MillWheel is a framework created to process data-in-motion with a low latency by providing fault tolerance, scalability and versatility without the drawbacks of previous systems. It was developed to handle Google's Zeitgeist pipeline, responsible for detecting trends in web queries. At the core of MillWheel, data is not grouped in mini-batches like it was in previous systems, being instead consumed as soon as it is available. MillWheel also introduced the concept of *watermarks* in order to define data that is delivered out-of-order. The open-source implementation of MillWheel is Apache Storm, which however does not have the same features: the former guarantees exactly-once delivery where the latter does not. The programming model of MillWheel went on to inspire Dataflow, which unifies batch and streaming processing.

The Dataflow programming model [2] unifies batch and streaming processing by separating the notion of data processing from the physical implementation of the engine that runs the job. In other words, a Dataflow program defines how data is processed, but whether this is done in batch, micro-batch, or streaming fashion is up to the engine that executes teh program. Google Cloud Platform implements the Dataflow programming model in its product, *Cloud Dataflow* (more details on the implementation in Section 3.1), which is based on FlumeJava and Millwheel. The reference API of Dataflow, Apache Beam [12], is very similar to FlumeJava.

A Dataflow pipeline consists of data flowing through multiple computation steps organized in a DAG. The data is first read through a *source*, goes through a number of computation steps, and is eventually written out in a *sink*. The source is a special `PTransform` that represents the input of the data e.g., a database or a distributed filesystem and produces a `PCollection` as output. It is similar to the read stage in MapReduce. Each "computation step" in the flow is a `PTransform`, responsible of applying a function on the input `PCollection` and producing a different output `PCollection`. The sink is a special `PTransform` that writes the data instead of producing an output. It is similar to the output stage in MapReduce. In general, data inside a `PCollection` is structured in (key, value) pairs as a $\texttt{PCollection}\langle \texttt{K}, \texttt{V} \rangle$ (where K and V denote the type) even though a key may not be required for some operations. There are two primitive `PTransforms` that are the building blocks of Dataflow programming:

- `ParDo`:

$$\texttt{PCollection}\langle \texttt{K}_1, \texttt{V}_1 \rangle \longrightarrow \texttt{PCollection}\langle \texttt{K}_2, \texttt{V}_2 \rangle$$

  This transform applies a function (called `DoFn`) in parallel to each element in the `PCollection`, producing 0 or more elements as output of a possibly different type as output. It is very similar to a `map` operation in the MapReduce world.

- `GroupByKey`:

$$\texttt{PCollection}\langle \texttt{K}, \texttt{V} \rangle \longrightarrow \texttt{PCollection}\langle \texttt{K}, \texttt{Iterable}\langle \texttt{V} \rangle \rangle$$

  This transform groups together records with the same key in a list, much like a `reduce` operation.

A key difference between Dataflow and MapReduce is that any number of `ParDo` and `GroupByKey` may appear in a single pipeline in any order.

The next section gives an overview on join algorithms and how they are applied in a parallel processing environment.

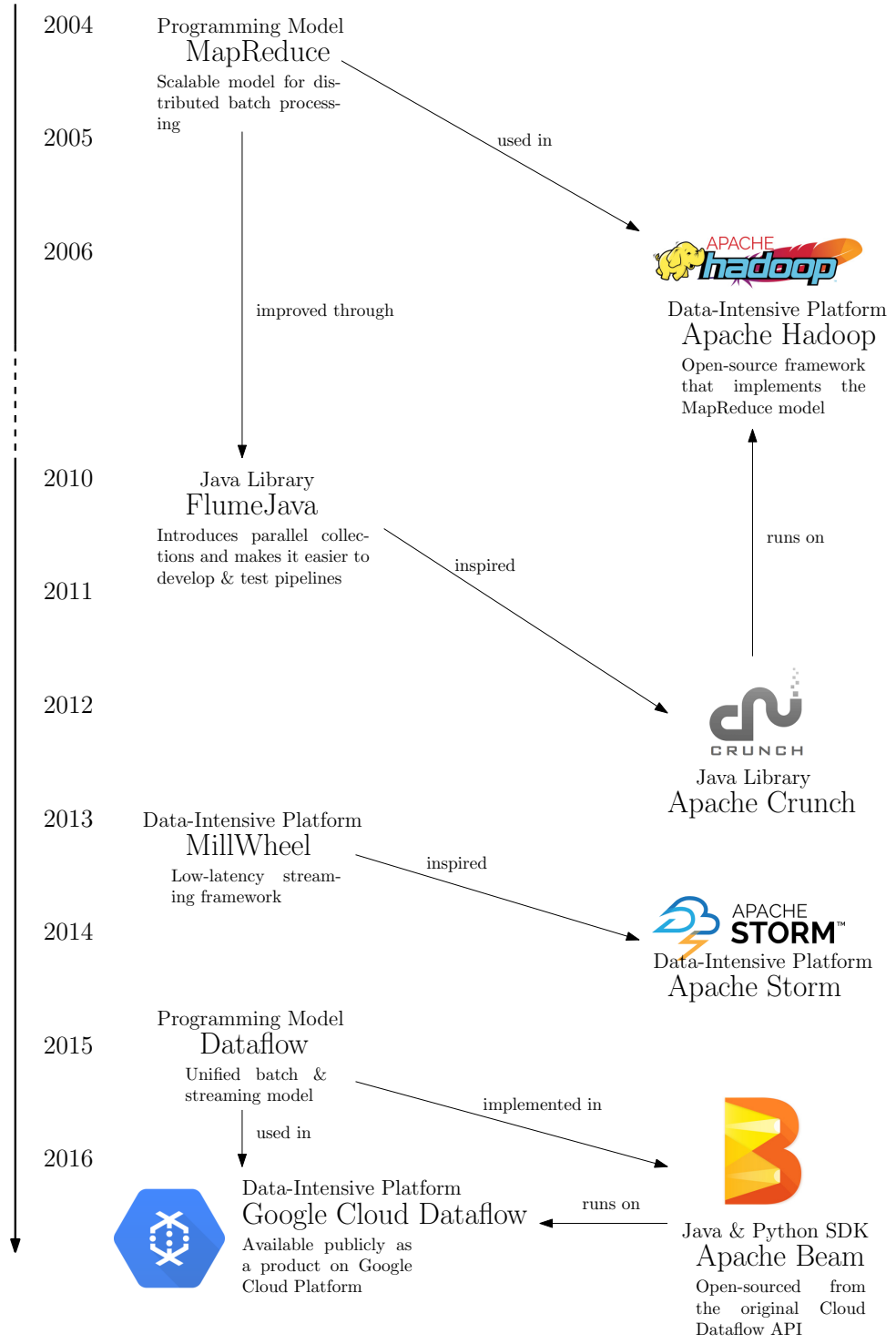


Figure 2.1: Timeline of key contributions to MapReduce that resulted in Dataflow.

2.2 Joins Overview

This section will briefly present join operations. Initially, join operations on a single machine will be presented: reasoning about these algorithms is important, as the design and the ideas behind these algorithms can still be applied to a parallel and distributed setting [16]. In fact, several distributed join algorithms work on partitioning the data such that records whose join keys would match end up in the same worker, which then applies one of the following algorithms. In describing these algorithms, big O notation is used to describe the number of IO operations required to compute the join of the two datasets R and S, which consist of collections of semi-structured data and may or may not fit in memory. The reasoning for analysing these algorithms from an IO perspective is due to the large scale of joins considered and because accessing disk (or network) is orders of magnitude slower than memory. Respectively, reading the R dataset requires p_R IOs and reading the S dataset requires p_S IOs. For example, if the tables were stored on disk, then p_R and p_S would represent the number of pages on disk. Apart from the naive nested loop join in $O(p_R p_S)$, there are two fundamental joining algorithms [17, 18] which will be explained below.

2.2.1 Join Algorithms

For the following two algorithms, the worst case scenario has the same complexity as the nested loop join, but it arises when *skewness* is present. For more information on skewness, see Section 2.3.

Hash join

This is the standard go-to join algorithm, with an example shown in Algorithm 1. Assuming without loss of generality that $|R| < |S|$ and it fits in memory, it works by creating a hash table of R (the build input), then querying it while reading S (the probe input). These two phases are respectively called the build phase and the probe phase. Note that a `MultiMap` is used because each key may be associated with multiple records. When the build input fits in memory, then the hash join computes the result in $O(p_R + p_S)$ IOs by simply reading R and then S. If the build input is too large to fit in main memory, both datasets can be partitioned using a hash function. This step is repeated recursively if resulting partitions from the build input still do not fit in memory by using different (orthogonal) hashing functions. In order to compute the result, Algorithm 1 is then called for pair of partitions from R and S. In this scenario, the number

Algorithm 1: Hash join between datasets R and S.

```
1 def HashJoin(R, S):
      Data: Datasets R and S, R fits in memory
      Result: R ⋈ S
2     H ← new HashMultiMap()
3     foreach r ∈ R do                          // build phase
4         H.put(r.key, r)
5     foreach s ∈ S do                          // probe phase
6         foreach r ∈ H.get(s.key) do
7             yield (r, s)
```

of IOs is still $O(p_R + p_S)$, as R and S are read once to write the partitions, then each pair of partitions is read once to compute the join. Consider the scenario in which some partitions of R do not fit in memory, for example due to a highly skewed key. In this situation, the build input partition are read in blocks, which are used build the hash table and join with the corresponding probe input partition. After processing one block, the hash table is emptied and the next block is processed. In practice, the algorithm works similarly to a block nested loop join. In an example worst-case scenario in which all keys are identical in R and it does not fit in memory, the algorithm still performs $O(p_R p_S)$ IOs (with lower constants as data is read in blocks). Optimizations such as hybrid hash join aim to improve the partitioning strategy and reduce the number of IOs, but asympototic complexity remains the same and a more detailed description is out of scope of this work. Picking "good" hash functions reduces the probability of the worst-case scenario.

Merge join

This algorithm is also divided in two phases. The first phase sorts the input tables by their join key, which are then joined with a merge join. A merge join works by scanning both tables in an interleaved fashion, while outputting records that satisfy the join condition.

Algorithm 2 shows an example implementation. The algorithm works by iterating over the two tables while comparing the join key. Records with the same join key are buffered into the respective buffer for R and S. For every key, the cross join of the two buffers is the output. If any buffer is empty, i.e. there are no records for that key value, the cross join results in an empty set.

If both tables fit in memory, then the running time is $O(p_R + p_S)$ by simply reading R and S. If the data does not fit in memory, external sorting techniques are used by writing runs of sorted data on disk. In this situation, the cost of sorting both tables is $O(p_r \log p_R + p_S \log p_S)$ [19]. In the average case, joining the sorted tables can be done in $O(p_R + p_S)$. Like a hash join, when skewness is present, then it might not possible to buffer the data and a block nested loop join is required in $O(p_R p_S)$. In general, as sorting the input data takes log-linear time, this method is not preferred to the alternative hash-based join. As a summary, merge join is $O(p_r \log p_R + p_S \log p_S)$ if sorting is needed and $O(p_R + p_S)$ if not, with better constants compared to hash join as no build step is needed.

Algorithm 2: Merge join between datasets R and S.

```
1  def MergeJoin(R, S):
       Input: Datasets R and S
       Output: R ⋈ S
2      sort(R)                    // External sort in O(p_R log p_R)
3      sort(S)                    // External sort in O(p_S log p_S)
4      rBuf ← ∅, sBuf ← ∅         // Buffers for R and S
5      i ← 0, j ← 0               // Index variable for R and S
6      while i < |R| and j < |S| do
7          if R[i].key < S[j].key then
8              i ← Advance(R, i, R[i].key, rBuf)
9          else if R[i].key > S[j].key then
10             j ← Advance(S, j, S[j].key, sBuf)
11         else                   // Note that R[i].key = S[j].key
12             i ← Advance(R, i, R[i].key, rBuf)
13             j ← Advance(S, j, R[i].key, sBuf)
14             foreach r ∈ rBuf do
15                 foreach s ∈ sBuf do
16                     yield (r, s)
17         rBuf ← ∅, sBuf ← ∅      // Clear buffers
18 def Advance(T, idx, k, B):
       /* Take records from T[idx] onwards and insert
          them into buffer B as long as their key
          is equal to k, then return idx          */
19     while idx < |T| and T[idx].key = k do
20         B ← B ∪ T[idx]
21         idx ← idx + 1
22     return idx
```

These two algorithms while being different also present a duality in several aspects. Graefe, Linville, and Shapiro [20] identify and catalogue these aspects, presenting advantages and disadvantages for the two algorithms. For example, both algorithms use recursion with an in-memory base case, but whereas the partitioning happens at a logical level for the former (based on hash value), it happens at a physical level for the latter (the position in the sorted data). In general however, hash joins perform significantly better than sort-merge joins because the recursion level is only determined by the *values* or distribution of the input data, whereas it is determined by the input *size* in the merge case. Many ideas from these joining algorithms can still be refactored and applied to parallel processing, and in particular the merge buckets optimization uses both hashing and sorting as will be explained in more detail in Chapter 4.

Following up on those findings in [20], it seems like the years have been kind to sort-merge joins. The current trend of bigger machines has narrowed the gap between sort-merge and hash joins [21, 22], with sort-merge join "likely to outperform hash join on upcoming chip multiprocessors" [22] due to better scalability potential of multi-core processing.

2.2.2 Distributed Joins

It is however different in distributed processing. In these environments, there is no shared memory and the cost of network communication dominates processing time. To understand why this is the case, there are three important factors that contribute to this:

1. **Serialization**: in order for the data to be communicated across the network, it must first be translated from memory objects to a format that can be transmitted over a connection. The serialized data is then used to recreate that same object in memory (the reverse process is called *deserialization*).

2. **Network IO**: when compared to hard disk IO, network operations are on average slower in throughput and setup time, despite this gap getting smaller and smaller over the years as data centers connection become faster. At the same time however, secondary memory access has also become faster with SSDs in place of HDDs.

3. **Stragglers**: "The wagon moves as fast as the slowest horse". In other words, a pipeline is not finished until all workers have finished processing. In situations where a particular worker is handling an oversized

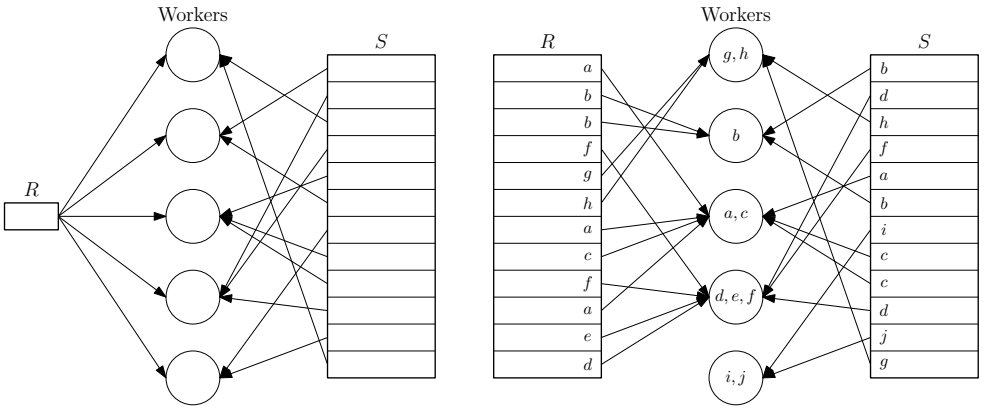
partition of data or there are failures, stragglers will delay completion time significantly.

In these algorithms, the big O notation describes the number of network IOs in terms of number of records exchanged between the workers, of which there are W in total. In Hadoop processing systems, two types of joins are built-in [23], visualized in Figure 2.2:

1. **Map-side join** (or broadcast join): Shown in Figure 2.2a. In this type of join, one of the datasets is very small and has $|R|$ records that can fit in main memory. This can be replicated across all the worker nodes ("broadcast") which can then use one of the algorithms presented above to perform the join operation (usually, hash join is used by creating an hash table on the replicated data). The name of this join comes from the fact that the Reduce phase is technically not needed at all: each worker can load the smaller table and join it with its share as part of the Map step. The number of exchanged records is $O(|R|W)$.

2. **Reduce-side join** (or shuffle join): Shown in Figure 2.2b. This is the standard join operation in a MapReduce setting. As the two table are too big to fit in memory, the data is partitioned according to its join key. The partitioning is achieved by using a hash function on the value of the key modulo the number of workers. The records are then shuffled to their respective worker and joined. The number of exchanged records is $O(|R| + |S|)$. Note that this is the default type of join used in Hadoop, in which it is also called *repartition join*.

When $|R|W < |R| + |S|$, the map-side join incurs in a smaller number of network IO. In Dataflow, both types of join are possible. The shuffle join is implemented through a `CoGroupByKey` operation, which uses the `GroupByKey` primitive. The map-side join can be implemented through the use of a `SideInput`, which represents an additional input that can be given to a `PTransform`. Side inputs are always broadcasted to all workers: the map-side join can be achieved as a `ParDo` in which each record from the larger dataset is matched against the smaller dataset passed as a `SideInput`.



(a) Map-side join.

(b) Reduce-side join. Each worker is responsible for a subset of the join keys.

Figure 2.2: Map- and reduce-side join between datasets R and S with $W = 5$ workers.

Join types

Depending on the comparator function used in the join to match rows from the tables, different types of joins are available. However, there are different semantics w.r.t. the output and which comparator function is used:

- **Inner join**: return only rows for which the comparator function is true.
- **Outer join**: retain rows even if the comparator function is false. If retaining the rows from the left/right table, it is a left/right outer join. Both rows can be retained, in which case it is a full outer join.
- **Equijoin**: if the comparator function is an equality between two columns, then it is called an equijoin.
- **Natural join**: It is an inner equijoin across all shared columns between the two tables.

In this work unless specified, join refers to the natural join, with a single shared column (the "join key"). It is also an equijoin, as the comparator function is equality of the shared key.

2.3 Skewness

In MapReduce and its evolutions, skewness refers to highly variable task runtimes which results in load imbalance between the worker machines [4]. Skewness is a major contributor to stragglers in parallel processing and it can affect both `map` or `reduce` operations (or their equivalents). Operations like `map` or `ParDo` do not affect how data is distributed throughout the cluster. These are also called *narrow dependencies* (as defined in Apache Spark), as opposed to *wide dependencies* like `reduce` or `GroupByKey` that cause data to be moved across workers. When shuffling data, additional sources of skewness are present which make wide dependencies more susceptible to skew. Skewness arises when a worker is overloaded by the data assigned to it, but determining its source is a hard task as there are multiple factors that can contribute to this phenomenon:

- **Data skew**: This is the traditional meaning of skew. Data skew refers to having "hot keys", a small set of keys that is shared by a large fraction of the dataset. This type of skew is applicable for `reduce` operations in which records are assigned to workers based on hash partitioning of the key. Hence, data with the same key would hash to the same value and be assigned to the same worker. Note that this side-effect is required in order to correctly compute join operations: if data with the same key would be distributed to different workers, the join output would not be correct.

- **Record skew**: Record skew refers to skewness in individual record sizes. Data that does not follow a strict structure may result in records having a different number of fields/sizes. Record skew may arise if the records contain fields of variable length, complex types, nullable fields, and more. Consider the following example:

  Example 1 *The dataset in the table below contains two records. All columns are typed as strings, encoded with an integer (four bytes) denoting the length of the string and then the string itself with one byte per character. The last three columns (`email`, `ip_address` and `country`) can be* NULL*, which is encoded as an empty byte.*

user_id	*email*	*ip_address*	*country*
6c68da	NULL	NULL	NULL
6c68da	*andnar@kth.se*	*130.237.28.40*	*SE*

In this example, the first record has a serialized size of 13 *bytes, whereas the second record has a serialized size of* 50*. Despite both records belonging to the same dataset, they suffer from record skew as the second record is more than three times the size of the first one.*

Record and data skew can combine their effects and create even greater skew when hot keys correlate with high record size (as opposed to non-hot keys with a small record size). Note that record skew can also affect `map` operations e.g., when encrypting data (bigger records take more time) or when the cardinality of the `map` varies with each record.

- **Partitioning skew** (or hash value skew): This skew is a side-effect of the use of a hash function in order to partition the data when shuffling. Consider the following example:

  Example 2 *Consider the task of partitioning some records over* $W = 3$ *workers. The hashed keys of the records are* $\mathcal{H} = \{0, 1, 2, 3, 4, 5, 7, 8, 11, 14, 17\}$*. A worker is assigned through a modulo function on the hash as* $w(h) = h \bmod W$*. The shares of each worker are*

  $$w_0 = \{0, 3\},$$
  $$w_1 = \{1, 4, 7\},$$
  $$w_2 = \{2, 5, 8, 11, 14, 17\}.$$

  In this example, the last worker is assigned more than half of the records. Note that all records have different join keys as they hash to different values: the skew in this scenario is independent of data skew.

When skewness, and hence stragglers, are present, the processing time is no longer dominated by network processing but by the processing speed of the slowest straggler. For example, consider the situation in which two datasets R and S contain only the same key. When performing a join, all the records will be shuffled to a single worker, who has to perform the join between all records locally. In such a situation, despite the amount of shuffled data being $O(|R| + |S|)$, the running time is more akin to $O(|R||S|)$ (the worst case scenario for an local join).

Handling skew is a hot topic in data-intensive computing research. The next section presents related work in how to handle and optimize joins when skewness is present in a MapReduce-like system.

2.4 Related Work

Early work by Lin et al. [24] shows how power-law distributions and in particular Zipf-like distributions lead to the creation of stragglers in parallel processing. In particular, they argue that this distribution can arise not only in input data, but also in intermediate data, imposing a limit on the degree of parallelism of certain tasks. As an example, the author shows how rearranging the input cuts running time in half for a pipeline that computed pairwise document similarity. The main takeaways of the author include difficulty in expressing efficient algorithms in MapReduce, and show how tight is the coupling between the model and the framework. Hence, awareness of the model (MapReduce) and the framework (Hadoop), is not only important but it is required in order to design correct and efficient algorithms.

Kwon et al. [4] present a general overview of skewness in MapReduce, including a list of different real-world applications which suffer from different types of skewness. For example, PageRank can suffer from record skew as some nodes in the computation have a significantly larger in-degree: processing of these nodes requires more CPU and memory and than other nodes. A series of best practices are provided in order to mitigate skew such as preaggregating data through *combiners* after the map phase, which compute a local aggregation of each worker before the reduce phase. Moreover, they suggest collecting properties of the data in previous processing before MapReduce in order to use different partitioning stragies. For example by collecting histograms on the join key, range partitioning can be used. Lastly, they suggest writing pipelines whose runtime do not depend on the data distribution, but only on the input size.

Blanas et al. [23] provide a systematic comparison of join algorithms on MapReduce. In addition to providing preprocessing tips to improve performance on the standard joins defined in Section 2.1, they present an implementation of semi-join on MapReduce. The semi-join is inspired from a real-world scenario similar to the one described in this work for the encryption pipeline. The intuition states that in a few hours worth of logs (events in this case), the number of unique users is actually in the few millions, hence joining with the complete dataset is not necessary. The semi-join preprocesses the data to determine the unique users and fetches only the corresponding part. The actual join happens as a map-side join as the few users can usually fit in memory.

Gufler et al. [25] define a cost model for handling skewed partitions of data and evaluate two solutions, *fine partitioning* and *dynamic fragmentation*. The cost of a partition is defined as a function of the total size and total number of

records of that partition. The first solution consists of partitioning data over a number of partitions $P > W$, where W is the number of workers. When scheduling tasks over the workers, the most expensive partition is assigned to the worker with lowest load, where the load is the total cost of all partitions assigned to that worker. The second solution splits larger partitions in smaller *fragments* at the map stage. When fragments of a partition are sent to different workers for the reduce stage, data from the corresponding partitions on other mappers must be replicated across all the reducers which are assigned the fragments. Dynamic fragmentation is also used in skew-adjusted SMB, in which a bucket is split into *shards*. The solutions are evaluated on synthetic data based on the Zipf distribution, lowering the standard deviation of worker load by more than 50% when data is heavily skewed.

With SkewTune [26], the authors present a drop-in replacement for the MapReduce implementation which automatically mitigates skew. At runtime, SkewTune determines stragglers and automatically repartitions data for which stragglers are responsible through range partitioning. A worker is defined straggler if the cost of repartitioning its task is smaller than half of the remaining time on that task. The cost of repartitioning (or repartitioning overhead) is an input parameter. They achieve an improvement of $4\times$ over the runtime of Hadoop 0.21 with SkewTune for some real-world applications. Whereas this solution is promising, it is tied to the MapReduce model and also requires having access to the implementation. In this work, modifying the implementation of the engine was not possible as the implementation of Cloud Dataflow is not available.

In [27], the authors tackle skewness in the context of join operations. Unlike the previous work, the authors do not modify the MapReduce framework, instead opting to handle skewness as part of their join algorithm, which is called MRFA-Join (MapReduce Frequency Adaptive Join). They achieve this by splitting the join in two MapReduce jobs. The first MapReduce job collects statistics about the data by computing histograms of the join key, whereas the second job uses the histograms to redistribute repeated (skewed) join attribute values randomly among a number of workers. This is achieved by pairing each record with a secondary key which represents the partition number for that join key value. The number of workers depends on the degree of skew: the more skewed is the key, the higher the number of workers. In order to evaluate their results, they generate records whose join key follows a Zipf's distribution for uniform values of shape parameter s from 0.0 to 1.0. The MRFA-join in the test scenario of a $400M \times 10M$ join achieves significant runtime improvements (from approximately 10000 seconds to 4000) and a large reduction in amount

of shuffled data compared to the join present in Hadoop (note that this join is available in the `contrib` module of Hadoop). In adition, the MRFA join is robust to all values of s tested, whereas the regular Hadoop join failed due to out of memory errors for $s > 0.5$.

Afrati and Ullman [3] focus on optimizing joins in MapReduce. In particular, the paper studies the case of multi-way join, i.e. with more than two relations at once. Instead of optimizing the order in which datasets are joined, the authors propose replicating data and performing the join in a single MapReduce job. As an example, the authors show how in a three-way join $R \bowtie S \bowtie T$, replicating data from R and T to multiple Reduce processes might lower the communication cost as opposed to communicating the results of the intermediate join. For example, in three-way join in which all tables have 10^7 records, each pair of records matches with probability 10^-5, and data is replicated using the described algorithm to 1000 Reduce processes, the optimized multi-way join in [3] reduces communication cost by a factor of 10. Despite the work focusing on multi-way joins, the idea of replicating data inspired several other works, which used it as a way to achieve straggler mitigation.

The previous skew-handling works are classified as *speculative execution* solutions by [28]. Whereas these solutions work at runtime by monitoring pipeline progress and mitigating stragglers, in [28] the authors propose using *proactive cloning*, through which tasks (in the Hadoop terminology) are replicated in a similar fashion to the previous work and the first available result for each clone group is used. The cloning algorithm determines a number of clones for each task based on the three input parameters: the number of tasks n, the cluster-wide straggler probability p, and the acceptable risk of stragglers ε. This approach limits the *job* straggler probability to at most ε. The authors also tackle the problem of intermediate data contention as a result of cloning. In order to mitigate the problem, they introduce *delay assignment* as an hybrid approach in which downstream consumers first wait a certain amount of time before reading intermediate data to get an exclusive copy. After that time, read with contention is used. They achieve improvements up to 46% in average running time on small jobs (between one to ten tasks), with that number decreasing to approximately 2% for large jobs (> 500 tasks). The use of replication in parallel processing had been studied before, but this work is the first that applies it to MapReduce-like systems. Note however, some limitations: computing the cluster-wide straggler probability is not always possible e.g., when using a managed service, and the solution modified the Hadoop implementation, which does not make it applicable in a broader scope. A similar approach of shard cloning is used in this work (see Section 4.4).

Chapter 3

Platforms

In order to properly evaluate the results of this work, this chapter describes context and engineering details of Cloud Dataflow, Google's service for data-intensive processing which implements the Dataflow model, and the platform used in this work. Section 3.2 provides a comparison of Cloud Dataflow with other data-intensive platforms, while Section 3.3 provides additional details on the event delivery infrastructure at Spotify.

3.1 Cloud Dataflow & Apache Beam

Google Cloud Dataflow is a fully-managed pay-per-use service for batch & streaming processing, available on Google Cloud Platform. It implements the Dataflow unified model as presented by the original paper [2]. Cloud Dataflow provides several features, such as pipeline visualization, logging, metrics, autoscaling, and more.

Figure 3.1 shows an example of a join operation in Cloud Dataflow. Each block corresponds to an operation, flowing top to bottom. In this case, the job is computing a join between two datasets. Each block reports the estimated time spent for that operation: note how the `CoGroupByKey` operation which shuffled data dominates all the others.

Figure 3.2 shows the job summary for previous join operation. It contains information such as job name, the Beam version used, the region in which the job executed, total elapsed time, and a plot of number of workers over time for purposes of autoscaling. Note that autoscaling was disabled for this job and the number of workers was fixed to 64.

Figure 3.3 shows metrics collected for the join operation, such as total number of CPUs, main and secondary memory (`PD`, persistent disk), and the

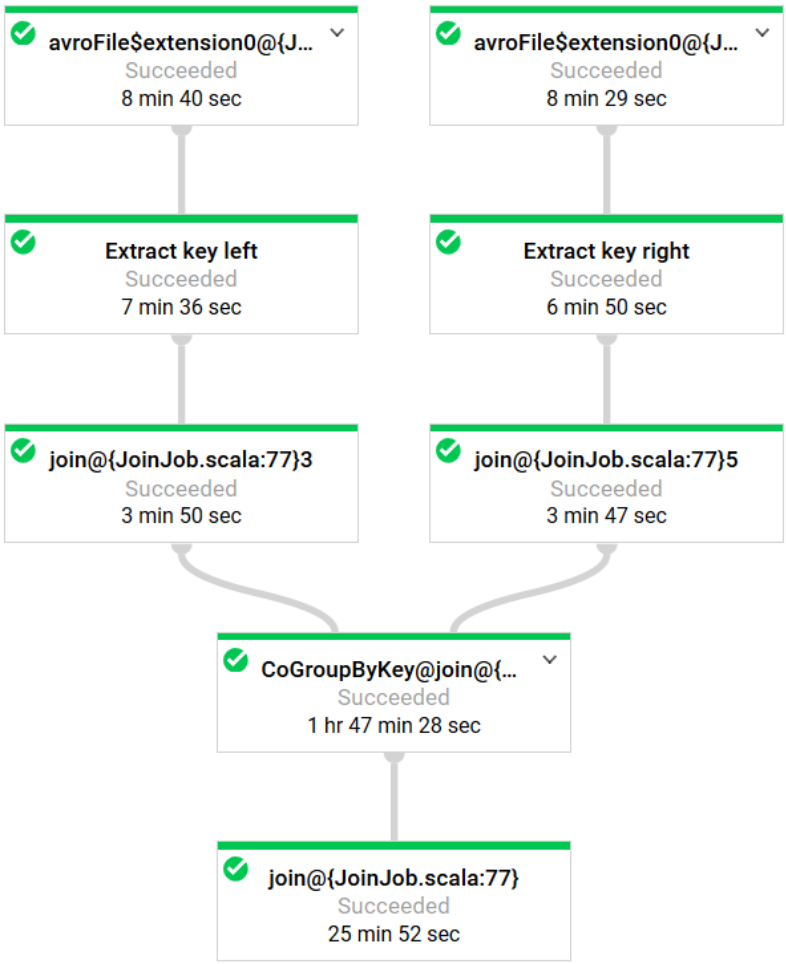


Figure 3.1: Visualization of the Dataflow graph for a join job in Cloud Dataflow.

respective CPU/memory/disk time used. The 64 workers in this job each had one CPU, 3.75 GB of memory and a 250 GB disk.

Users do not have access to Dataflow itself, and instead submit pipelines written with Apache Beam [12] SDKs, available in Java, Python and Go. The Apache Beam SDKs have a very similar APIs to FlumeJava, defining operations in terms of `PTransforms` on `PCollections`, but has two additional levels of abstraction. Firstly, Beam pipelines are *runner*-independant. A runner is defined as the back-end system that will execute that job. In this work, Beam uses the Cloud Dataflow runner, but several other runners are available that can translate the Beam job to other data-intensive platforms such as Spark and Flink. Secondly, in the unified model of Dataflow, the same Beam pipeline can be run in both streaming and batch mode. In other words, the user writing

an Apache Beam pipeline does not need to be aware of which runner and which semantics (batch or streaming) the pipeline will be used in, which has advantages:

1. **Portability**: As the same pipeline can be used for multiple backends and for both batch and streaming use-case, this greatly increases code reusability. As Dataflow implies that "batch is a special case of streaming", Beam pipelines make can make the switch to streaming easier by reusing the same code.

2. **Low barrier of entry**: As execution details are unknown and abstracted away, writing pipelines in Apache Beam has a low barrier of entry as users can get started more easily as fewer concepts need to be learned.

3. **Testability**: Beam comes with a `DirectRunner`, which executes a pipeline locally. This makes it much easier to prototype & test pipelines before running it on a cluster.

At the same time, it also presents disadvantages:

1. **Runner abstraction**: Whereas runners abstract away complicated details, the Beam model makes it impossible to use runner-specific optimizations or mix runner and Beam code. Similarly, it is not possible to implement low-level optimizations that operate on how a runner works, as which runner will execute the pipeline is unknown at compile time. In other words, how a job is executed is up to the runner, which is not decided until the pipeline is actually executing. For example, this implies that the notion and number of workers can be different if the pipeline is running locally on a single machine, or on a Spark or Cloud Dataflow cluster. In other words, this means that it is not known how tasks will be split between each worker or having fine-grained code that is aware of specific runner configurations.

2. **Ordering**: From an API perspective, `PCollections` are considered as unordered bag of items, even if the underlying runner might support ordering in their implementations (for example this is the case for Spark and RDDs).

For purposes of fault tolerance and scalability, data in Beam is processed as part of bundles. How bundles are created is runner-dependant (note that this notion is different than the notion of windows in streaming). Bundles are used in order to determine when and how data should be persisted: this allows

the runner to implement fault tolerance per-bundle. In other words, if one or multiple records fail processing in a bundle, only that bundle will be repeated. Note that the failed bundle might be retried by different workers than the ones that processed it originally. By default for batch processing pipelines in Cloud Dataflow, a bundle is retried four times before the whole pipeline is deemed to have failed.

In addition to the primitives of the Dataflow model, Apache Beam presents four additional core `PTransform`:

1. `CoGroupByKey`: This operation performs a relational join operation between two or more input `PCollections`. It can be seen in action in Figure 3.1.

2. `Combine`: This operation is used to aggregate data in an efficient way. The user provides an associative, commutative function that contains the logic for combining the values. The function may be evaluated multiple times on each value in order to create partial aggregations that reduce the data shuffled. The idea comes from *combiners* in MapReduce, which perform an aggregation locally between the `map` and `reduce` stages. As an example, finding the maximum value of a `PCollection` of integers can be done through a `Combine`, as the `max` function is associative and commutative.

3. `Flatten`: This operation computes the union of multiple `PCollection`.

4. `Partition`: Splits a `PCollection` into multiple `PCollection`.

Submitting a Beam pipeline to Cloud Dataflow triggers a series of steps:

1. The compiled pipeline code is uploaded and saved to a temporary folder.

2. The Cloud Dataflow service constructs the Dataflow graph and applies optimizations (this is called the **Graph Construction Stage**). An example optimization is `ParDo` fusion, which is similar to how FlumeJava combined multiple MapReduce stages together. A similar optimization happens on `Combine` operations.

3. Dataflow runs inside containers on the workers. The input data is split into bundles and assigned to the workers which start processing it.

4. The Cloud Dataflow service monitors pipeline execution. If autoscaling is enabled, the current throughput is used to increase or decrease the number of workers.

The Cloud Dataflow runner may create temporary files during execution for a variety of reasons. For example, this can happen to persist data, free memory from workers for other processing steps, or because the data does not fit in memory and spills to disk. Much like network communication, these steps require serialization and contribute to slow down the whole pipeline.

Job summary

Job name	joinjob-s020-1558282542-6d882b35
Job ID	2019-05-19_09_16_09-15493170235786862502
Region	europe-west1
Job status	Succeeded
SDK version	Apache Beam SDK for Java 2.11.0
Job type	Batch
Start time	19 May 2019, 18:16:10
Elapsed time	7 min 2 sec
Encryption type	Google-managed key

Auto-scaling

Workers	0
Current state	Worker pool stopped.

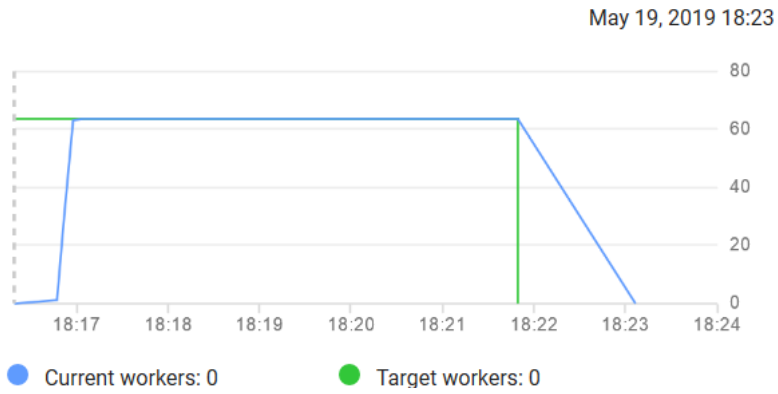


Figure 3.2: The job summary for an example join job.

Resource metrics

Current vCPUs	64
Total vCPU time	5.293 vCPU hr
Current memory	240 GB
Total memory time	19.849 GB hr
Current PD	15.63 TB
Total PD time	1,323.257 GB hr
Current SSD PD	0 B
Total SSD PD time	0 GB hr

Figure 3.3: Collected metrics for an example join job.

3.2 Platform Comparison

This section contains a overview of the functionality of other popular data-intensive platforms. A summary is presented in Table 3.1. The year refers to the first public release under that name: several projects were previously worked on under different names or the name changed when the project was donated to the Apache Software Foundation. The measure of popularity in this case is given by the number of stars on GitHub for open-source platforms. The following list provides a small description for each framework:

1. **Apache Hadoop**: See Section 2.1 for a description of MapReduce and Hadoop.

2. **Apache Storm**: Storm is a realtime framework for streaming data. Its programming model defines a topology (a DAG) organized in *spouts*, which identify a data source, and *bolts*, which process the data coming from the sources. It is fault-tolerant, scalable and guarantees at-least-once processing.

3. **Apache Spark**: Spark is arguably the most popular data-intensive platform. Its programming model is based on Resilient Distributed Datasets (RDD), which are similar to FlumeJava's `PCollection`. It was originally created as a batch processing framework but has since expanded to have structured data processing (Spark SQL), streaming (Spark Streaming), machine learning support (MLlib) and graph analysis (GraphX). On the performance side, Spark greatly improved on Hadoop MapReduce by e.g., keeping data in memory instead of writing it to disk after each intermediate step. In order to still achieve fault tolerance, operations on RDDs are logged in a dependancy DAG called *lineage graph*. Spark Streaming, despite the name, is not actually a streaming engine but splits data in micro-batches. The difference is subtle: stream processing handles data as soon as it is available, whereas a micro-batch framework introduces delays in order to wait for that micro-batch to be "complete", increasing latency. The original creators of Spark founded the company Databricks, which offers an enterprise analytics platform powered by Spark. A Spark runner is available for Beam.

4. **Apache Flink**: Apache Flink is a framework for stateful computations over data streams. These data streams can be bounded or unbounded: in other words batch processing is viewed as a special case of streaming in which the stream is eventually finished. Flink also implements

Table 3.1: Comparison of data-intensive platforms.

Name	Year	Programming Model	Processing Model	Open-Source	Relative Popularity
Apache Hadoop	2006	MapReduce	Batch	Yes	9108
Apache Storm	2011	Spout & Bolt Topology	Streaming	Yes	5695
Apache Spark	2012	RDD	Batch & Micro-batch	Yes	22068
Apache Flink	2014	Dataflow	Batch & Streaming	Yes	8873
Cloud Dataflow	2015	Dataflow	Batch & Streaming	No	*n/a*

the dataflow model, and it is no surprise that it is the most supported runner alongside Cloud Dataflow. The original creators of Flink created Data Artisans (now Ververica), which similarly offers commercialized analytics platform on top of Flink.

Spotify's event delivery infrastructure make large of use Cloud Dataflow, as is detailed in the next section.

3.3 Event Delivery Infrastructure

The event delivery infrastructure at Spotify (visualized in Figure 3.4) is responsible for receiving, preprocessing and providing the data to other consumers inside of Spotify, e.g. teams responsible for producing song recommendations or that calculate royalties [29]. Each event is received through Google Pub/Sub, a message broker similar to Apache Kafka. Most of Spotify's event delivery infrastructure uses Cloud Dataflow in batch mode (instead of streaming). There are a number of historical, performance and business reasons for this. Spotify at one time had the biggest Hadoop cluster in Europe, with approximately 2500 nodes. It was retired in 2018, but much of its legacy still remains and influences the current infrastructure.

The preprocessing part of the event delivery infrastructure consists of several ETL workflows that operate on events by reading and writing hourly-partitioned files in Avro format on Google Cloud Storage. The workflows are written using the Apache Beam SDK or using Scio [30], a Scala SDK

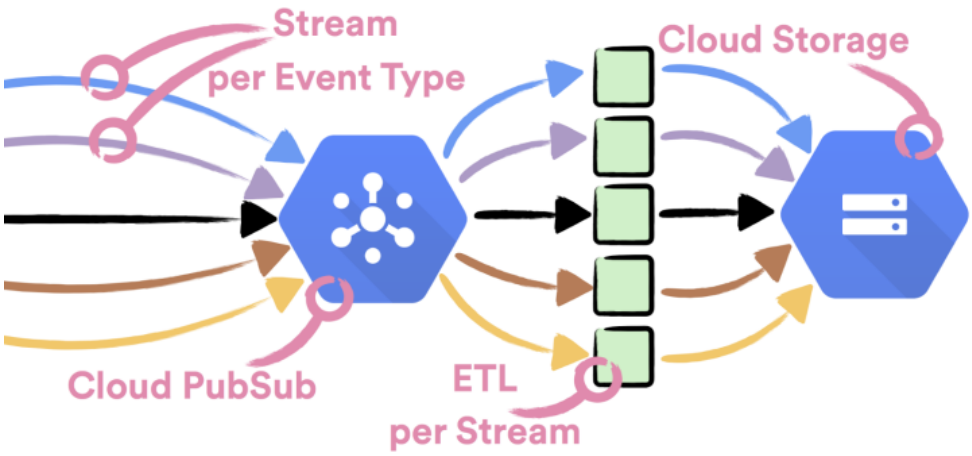


Figure 3.4: Diagram of Spotify's event delivery infrastructure.

for Apache Beam also developed at Spotify. The preprocessing has two main goals:

1. Data deduplication: as events travel from the client back to the infrastructure, non-optimal network conditions or other faults can affect the quality of data received. Hence, some events may be duplicated as clients attempt multiple retries at sending the data. Moreover, the messaging broker Pub/Sub may introduce duplicates as it guarantees at-least-once delivery (as opposed to exactly once).

2. Data encryption: all data containing sensitive user information that is persisted at Spotify is encrypted. Downstream consumers of the encrypted data inside Spotify need to first decrypt the data before accessing it. In order to access data, each team undergoes approval from the Legal and Data Protection offices.

Both of these use cases involve joining with large datasets: the deduplication pipeline joins with a windowed index of previously seen events in that time window. The larger is the deduplication window, the larger the cost of performing the join. The encryption pipeline is responsible for encrypting all sensitive data that is stored. It does so by joining with a large table of user encryption keys (each user has a unique key). The same join operation needs to be repeated in order to decrypt the data and process it. The use case of encryption and decryption is the focus of this work and the solution presented in the next chapter.

Chapter 4

Solution

The solution presented in this work is called *SMB* (SMB). The technique combines several ideas from how joins are implemented in relational databases and it is also available as an experimental feature in Apache Hive, a data warehouse that allows SQL-like queries running on Hadoop. The main idea of sort-merge buckets consists of preprocessing the data in a such a way that later joins can reap the benefits. Before describing the solution in detail, Section 4.1 describes motivates the methodology used in this work. Section 4.2 briefly describes challenges in the context of the thesis project, and the rationale for using SMB. Afterwards, Section 4.3 describes the initial solution and Section 4.4 describes how to extend SMB to overcome its limitations and tackle skewness.

4.1 Methodology

Evaluating data-intensive platforms is a hard task. Most of the works presented in Section 2.4 use an empirical approach in order to evaluate their solutions. This choice of methodology is explained by the complexity of these platforms: the amount of components or factors (network, failures, slow provisioning, ...) that can contribute to the evaluation of a solution in a parallel-processing scenario make it hard to create a formal model. Instead, most works adopt an empirical methodology in which metrics are compared between the experimental solution and a baseline. This chapter describes why an empirical methodology is chosen for this work, in addition to presenting several engineering details for Cloud Dataflow, the data-intensive platform used for this work. These engineering details are important in order to properly understand the context of the empirical evaluation.

Whereas a theoretical model of a complex system like data-intensive platforms provides essential results in terms of asymptotic complexity, correctness, and more, it often abstracts several important components (e.g., network, hardware, ...) that are key to the evaluation of a platform or a particular solution. Moreover, two algorithms with the same asymptotic notation might have very different constants and performance in a real world scenario. The empirical methodology, which tests hypotheses through experiments, has the advantage that no formal analysis is required and results take into account the whole platform, but makes reproducibility difficult for several reasons. When evaluating performance of a solution against previous research, it is often not possible to reproduce exactly the same conditions. This applies not only to cluster configuration, but also input data and possibly library versions. For example, it is important to demonstrate that the benefits of a particular solution are actually because of that solution and not because of performance improvements due to a library upgrade. In other research fields such as computer vision or machine learning, there exist datasets which are often used as benchmark in order to evaluate the proposed model or solution. Due to the wide scope of MapReduce & derived systems, no benchmarks are available. In most works where an empirical evaluation is used, it is hence required to setup a baseline and a experimental scenario to test the solution while all other factors are kept fixed.

An empirical solution is compared against its baseline through a variety of metrics. The most commonly used are as follows:

1. **Job duration**: measures the total elapsed time of a job from start to finish.

2. **CPU time**: measures the total processing time. For example if two workers with one CPU each worked on a job for half an hour, the total CPU time is one hour.

3. **Network IO**: the total amount of bytes exchanged via the network for the job.

4.2 Challenges

The engineering context as described in Chapter 3 presents several challenges in designing & evaluating possible solutions.

1. **Runner-independent**: Apache Beam's runner abstraction makes it so that the proposed solution solution can not be tied to a specific implementation. For example, this means that there is no notion of ordering for

records in a `PCollection` and it is not possible to retrigger straggling tasks from a scheduler. On the other hand however, the runner abstraction means that the solution will work on all of Beam's runners, which possibly makes it broadly applicable for a large variety of data-intensive platforms and both batch and streaming processing. Due to this challenge, solutions and related work that modified platform implementations are not directly applicable, and a proactive approach is instead required.

2. **Compatibility**: The designed solution should be backwards-compatible. For example, for solutions that use preprocessing steps, the data should not be modified in such a way that regular join operations are no longer possible. Similarly, the data should still be usable for non-join operations even after the preprocessing step. This challenge is motivated by practical considerations: backwards-compatibility makes it easier to adopt the solution, especially at large scale. Scale here refers both to the massive amount of data, but also the amount of beneficiaries of the solution. In Spotify's example, a broadly compatible and easy to apply solution is benefitted by the many consumers of event data, which need to decrypt it every time before use. Consequently, it is also important for the solution to be simple and require minimum user input in order to maximize transparency.

3. **Skewness & Scalability**: the solution should handle both skewed and non-skewed data, and be scalable. This is also motivated by a series of practical considerations: malicious users could generate millions of events, duplicate events can be a problem due to non-optimal network conditions or because of Pub/Sub's at-least-once guarantee, and the hourly partitions can vary wildly in size & skew throughout the day. In practice, some hourly partitions having more than 10 times the events of other partitions (e.g., when comparing afternoon and night hours).

4. **Minimum overhead**: This challenge is twofold. First, it refers to the *repeated* keyword of the described join scenario. When the join is not repeated, the solution should present as small as possible an overhead over a single join operation. Secondly and connected to the previous point, the overhead should be minimum when data is not skewed.

As a speculative execution approach (as defined in [28]) is not possible due to the runner abstraction, a proactive, two-phase approach is used. The SMB solution hence has two phases: the first *bucketing* phase partitions the data into sorted buckets. The second phase joins two datasets that are bucketed and

sorted. Intuitively, this solution works like existing work on skewness in which the first phase collects statistics on the data, and then uses a custom partitioning strategy to prevent stragglers. In presenting the solution, the research questions will be addressed in order: first, SMB will be introduced to address the *repeated* join problem. Then, the solution is extended to handle skew and additional concerns.

4.3 SMB Join

In order to optimize the repeated join scenario, a merge join can be used as opposed to a hash join approach. Recall that, if the data is already sorted, a merge join has better performance than the hash join as no build phase is needed. However, there are some challenges in producing sorted data. Firstly, `PCollections` are by definition unordered bag of items. Even if they were, achieving a global ordering on the data is not possible: expressing iterative algorithms like sorting is not supported by the programming model.

When computing a join in parallel processing systems like Cloud Dataflow, shuffling dominates processing times. Data is shuffled in order to parallelize the work: however, this requires that data that would join ends up in the same worker in order to compute correct results. Another way of looking at the problem would be determining the join set for each record. In the database world, an index can be used for this purpose: it allows quick lookup operations to achieve fast data retrieval. Partitioning data can be thought of as a very rudimentary index with no state. Looking up the index consists of computing the hash of the join key, which returns the "location" (the worker) that holds records with the same key. Every time a join (and hence, a shuffle) is performed, the data is repartitioned to the correct worker.

In order to still use a merge join, we can relax the global ordering requirement by requiring a local order instead: the data can be partitioned in such a way that each partition, or bucket, is locally sorted. Essentially, SMB consists of a merge join over bucketed data, which has been preprocessed in a such a way that the "index" becomes implicit in the structure of the data. Hence, it is possible to compute the join of two datasets preprocessed in the same way by simply performing a merge join operation over each pair of corresponding buckets.

The SMB join procedure consists of two phases. The first phase is *bucketing*, visualized in Listing 3. The bucketing `PTransform` is a composite transform which uses three other `PTransforms` invoked through method chaining. The bucketing phase is composed of two subphases:

Listing 3: Bucketing phase for SMB.

```
1 def BucketingPhase(input, B):
      Data: input is a PCollection⟨V⟩, with V denoting the type of the
            records inside the PCollection. B is the number of buckets.
      Result: PCollection⟨Bucket⟨V⟩⟩
2     return input
3        .apply(ParDo.of(ExtractBucketKeyFn(B)))
         /* Returns a PCollection⟨K, V⟩, where the key
            K is the bucket key determined through
            Equation (4.1) over B buckets.          */
4        .apply(GroupByKey.create())
         /* Returns a PCollection⟨K, Iterable⟨V⟩⟩    */
5        .apply(ParDo.of(SortBucketFns()))
         /* Locally sorts the records in an
            iterable by their join key, creating a
            Bucket⟨V⟩ and returning a
            PCollection⟨Bucket⟨V⟩⟩.                  */
```

1. **Partitioning** (lines 3, 4): the data is first partitioned over a number of buckets B, through a hashing function h on each record. Whereas $x.key$ refers to the join key for record x, the bucket key or bucket id $b(x)$ is determined as

$$b(x) = h(x.\texttt{key}) \bmod B. \tag{4.1}$$

   Note that partitioning happens as an explicit operation in Dataflow, and is not an implementation detail of the runner. It works by extracting the bucketing key of each record and then through a `GroupByKey`, which groups records by their bucketing key into an iterable. In this implementation, the hash function h is MurmurHash3 [31], which is the default hash used in Hadoop. This hash function is used because of its excellent performance and good distribution. The `GroupByKey` operation in this step is the only shuffle operation in whole pipeline.

2. **Sorting** (line 5): Recall that a `PCollection` does not have a notion of ordering for its contents. However, the output from the previous step is a `PCollection` of buckets. In this scenario, the records inside each bucket are being sorted by their join key, and each bucket in the `PCollection` is located on one worker only: hence, it is possible to perform a local sort of each bucket in parallel. The sorting is achieved through a merge

sort, falling back to an external merge sort if data does not fit in memory.

At the end of the bucketing phase, each bucket is written to an Avro file with its bucket id, the total number of buckets, the hash function used, and the join key as metadata. Users can inspect the metadata to know whether the data has been bucketed and and for which field it is sorted, and hence whether it can be used for a SMB join. Alternatives approaches to storing the metadata are possible, such as storing it in a separate location from the data or in a different file with the data. In practice, this preprocessing step consists only of modifying the layout of data by sorting it, which makes it compatible with existing pipelines for users that are not aware of the underlying structure of the data. Reordering the data has the additional advantage that data compression can achieve a much better compression ratio, as data with the same join key ends up being stored in an uninterrupted block. This is significant when several fields of data correlate with the join key: for example when the join key is the user id, it is reasonable to expect that fields like country, IP address, device, and more do not change in an hourly window for each user.

The preprocessed data can be joined with another *compatible* dataset. Compatibility between two datasets is achieved when the following conditions match:

1. **Number of buckets**: The two datasets must be partitioned over the same number of buckets. This ensures that bucketing key for two records that have the same join key is the same.

2. **Hash function**: Using the same hash function for both datasets ensures that the same join key hashes to the same bucketing key.

3. **Join key**: Both datasets must be partitioned & sorted on the key that is used as join key. If the data is sorted by a different key, it can not be used for a merge join and re-bucketing is necessary.

Note that bucketing both inputs is not additional requirement when compared to normal join operations: when performing these, both datasets are shuffled to the corresponding worker. Similarly, both datasets need to go through the bucketing phase.

The second phase of SMB is *joining*, visualized in Listing 4. This operation is executed as the first `PTransform` of a different pipeline. The start of a pipeline in Apache Beam is represented as `PBegin`, which can be thought of as a special, empty `PCollection`. The joining phase also consists of two subphases:

Listing 4: Joining phase for SMB.

```
1 def JoiningPhase(rPath, sPath):
      Data: rPath and sPath represent the location where the bucketed
            data for the datasets R and S is stored.
      Result: PCollection⟨V_R, V_S⟩, with V_R representing the records of
              R and respectively for S.
2     return PBegin
3        .apply(ParDo.of(ResolveBucketingFn(rPath, sPath)))
         /* Returns a PCollection⟨Bucket⟨V_R⟩, Bucket⟨V_S⟩⟩,
            with each tuple representing a matching
            pair of buckets.  Two buckets match if
            they have the same bucket id.  The
            records inside each bucket are not read
            yet.                                      */
4        .apply(ParDo.of(MergeJoinFn()))
         /* Performs a merge join as detailed in
            Algorithm 2 over each pair of buckets,
            returning a PCollection⟨V_R, V_S⟩.        */
```

1. **Bucketing Resolution** (line 3): this step ensures that the input datasets are compatible and generates the corresponding pair of buckets. In the implementation used in this work, it is executed on a single worker as the number of buckets is very small and it is a very fast operation. It works by expanding the file patterns for R and S to match each bucket, from which the metadata is extracted. The two lists of metadata are iterated to determine compatibility and the pairs of matching buckets.

2. **Merge join** (line 4): each pair of buckets is joined using a merge join operation as described in Section 2.2.

When two datasets are compatible, the result of the sort merge bucket join is correct. This is due to the fact that records that have the same join key will hash to the same value, which is assigned to the same bucket. In other words, no tuples that have the same join key will be in different buckets. The steps in order to compute a SMB join are summarized in Figure 4.1. The two datasets R and S are first bucketed by partitioning and sorting on their join key, then each corresponding bucket is joined through a merge join operation. The letters represent values of the join key. The total amount of output records from this join is 14: 7 from the first bucket ($6 + 1$ pairs for keys a and d), 3 from the

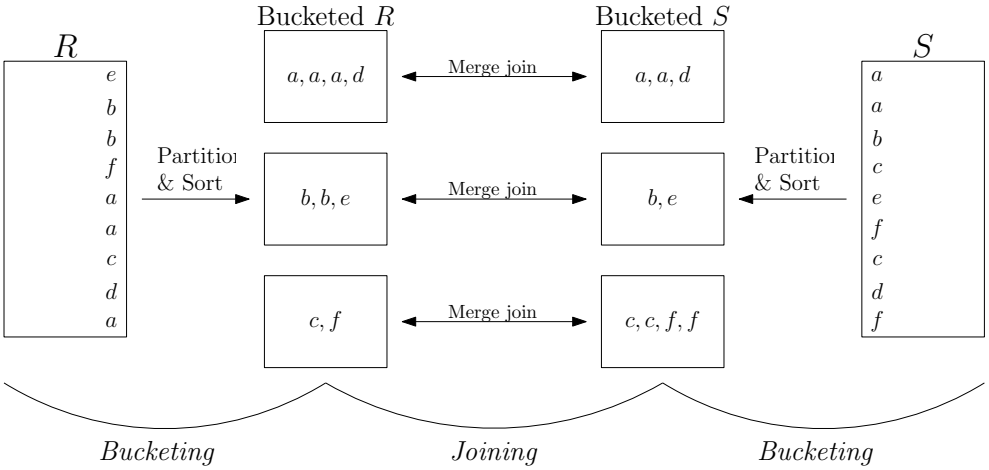


Figure 4.1: High level overview of SMB join.

second bucket ($2 + 1$ for keys b and e) and 4 from the third bucket ($2 + 2$ for keys c and f).

Figure 4.2 show what the bucketing looks like in Cloud Dataflow. The operations correspond to:

1. Read data stored in Avro format.
2. Create (key, value) pairs as in line 3 of Algorithm 3.
3. `GroupByKey` (line 4).
4. Sort iterables (line 5).
5. Second part of the sorting operation in line 5, implemented as an additional `PTransform`. Wraps the pairs $\langle K, \texttt{Iterable}\langle\texttt{V}\rangle\rangle$ to a $\texttt{Bucket}\langle\texttt{V}\rangle$ with the corresponding metadata.
6. Write out the data in Avro format, one file for each bucket.

Figure 4.3 show what the bucketing looks like in Cloud Dataflow. The operations correspond to:

1. Match list of files on the left and right side.
2. Read metadata from each file.
3. `CoGroupByKey` with a null key in order to group the metadata on one worker, responsible for bucketing resolution.

4. Create pairs of buckets to merge join.
5. Reshuffle the pairs of buckets to all the workers.
6. Read and merge join all the buckets, creating groups of records from the two datasets for each key.
7. Compute an inner join, i.e. emit only groups that have records with that key for both tables.

The SMB join as such can be used to optimize the repeated join scenario. The encryption pipeline can produce bucketed data as part of the bucketing phase, enabling downstream consumers to decrypt it through the joining phase. The bottleneck of shuffling is performed only through the `GroupByKey` in the bucketing phase, which is performed only once for any number of joins. As part of using Apache Beam, the solution is compatible with all the supported runners and more: SMB join, as presented here, simply consists of reordering data and a joining step that can be implemented in any `map`-like operation. A similar approach can be used in MapReduce-like platforms and all its evolutions.

While SMB join is effective in optimizing repeated joins (see Chapter 5 for evaluation), there are a few limitations. First of all, datasets that are bucketed need to be compatible. In addition, this type of join introduces added complexity in the form of choosing the number of buckets. Intuitively, if $B = 1$, all the data will be grouped on a single worker, which is unfeasible for datasets that do not fit in memory. As the number of buckets increases, the data is redistributed in multiple buckets. Due to the log-linear term in sorting, a higher overall number of buckets means a lower overall sorting time as can be seen in the following example:

Example 3 *Consider bucketing a dataset with n records over B buckets. Each bucket receives an equal n/B share of records and fits in memory. The total cost for sorting B buckets is*

$$B \cdot O\left(\frac{n}{B} \log \frac{n}{B}\right) = O\left(n \log \frac{n}{B}\right).$$

As the number of buckets increases and gets closer to n, bucketing becomes less advantageous as the algorithm regresses to a repartition join. In the same setting as Example 3 an extreme case of $B = n$ would result in no sorting at all as each bucket contains one record which is already "sorted".

Good values for number of buckets hence lie in the middle. A practical approach to determining the number of buckets would be picking a number such

that buckets fit in memory of each worker: this avoids overloading a worker due to the `GroupByKey` operation and it avoids having to fallback to slower external sort operations while at the same time maintaining the benefits of sorting. However, determining "fit in memory" criteria presents some problems:

1. **Input size**: It is necessary to know the size of input data. This can be estimated or can be computed as another Dataflow job.

2. **Runner abstraction**: In order to determine if a bucket fits in the memory of workers, runner-specific information such as number of workers and memory of each worker is needed. This can not be determined at runtime as per the runner abstraction, hence the programmer must be aware of these values. This represents a failure with regards to the first challenge in Section 4.2.

3. **Data distribution**: An approach that simply divides the total dataset size over the available memory assumes that buckets contain an equal share of records. However, such a distribution is unlikely due to skewness: as a result, some buckets will spill to persistent storage.

While the SMB join presents some advantages, it also raises some issues. Most importantly, the last point mentioned above leads the work to the second research question: addressing skewness.

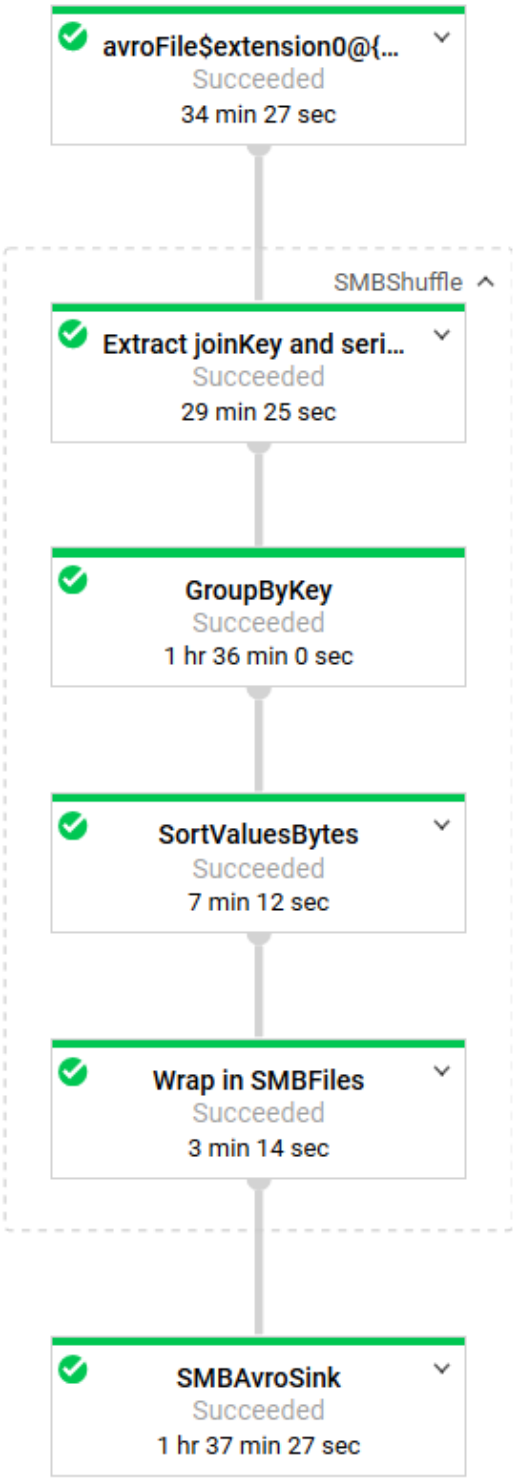


Figure 4.2: Bucketing pipeline in Cloud Dataflow.

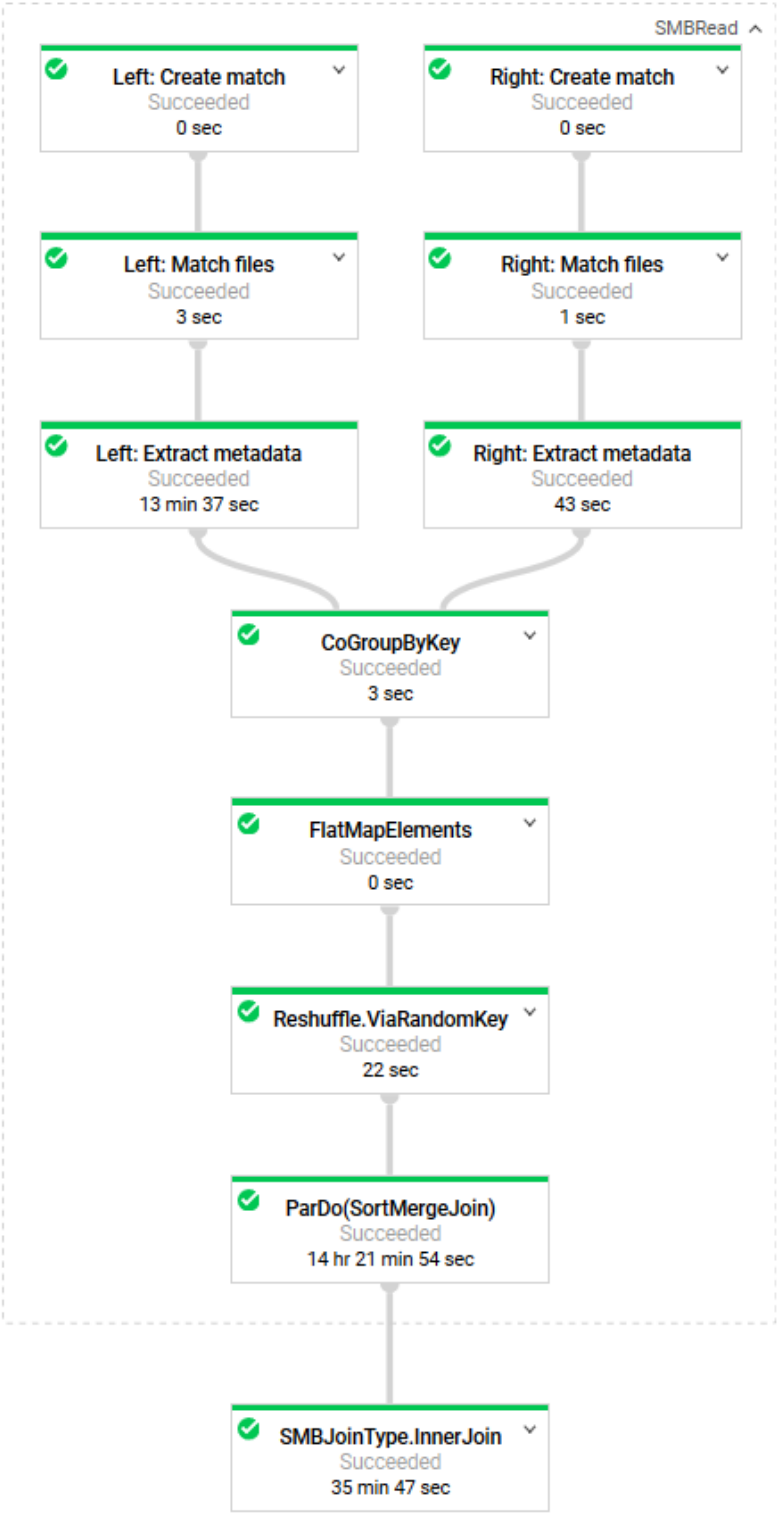


Figure 4.3: Joining pipeline in Cloud Dataflow.

4.4 Skew-Adjusted SMB Join

As elaborated on in the previous sections, skewness is a major problem in MapReduce-like processing systems and also arises in SMB joins. In particular when the data is skewed, the bucketing operation is affected due to the `GroupByKey` and sorting operations. In addition to skewness, an additional issue lies in the notion of dataset compatibility, which is limits flexibility when preprocessing data and joining.

In order to address these problems this section presents adjustments in order make the solution robust to robust and overcome the shortcomings described.

The first adjustment deals with *bucket size* and dataset compatibility. Recall that two bucketed datasets can be joined, i.e. are compatible, if they have the same number of buckets and were bucketed using the same hash function on the join key. However, any two datasets can be joined, independently of the number of buckets, by replicating buckets. Consider two datasets R and S, which have been bucketed over B_R and B_S buckets respectively. As the two datasets have different number of buckets, there is no correspondence between keys of the two different datasets as the partitioning function had a different modulus (the second operator). A naive approach of joining the data would be computing the merge join between all pairs of buckets: the results are correct as all pairs of buckets are joined and hence all matching records would be joined. This results in each bucket being replicated a number of times that is equal to the number of buckets on the other side, computing a total of $B_R B_S$ merge joins. However, it is not necessary to compute the merge join for all pairs of buckets. Using the distributive property of the modulo operation, for integers x, a, b we have that

$$\begin{aligned}(x \bmod ab) \bmod a &= (x \bmod a) \bmod a + (a \cdot (x/a \bmod b)) \bmod a \\ &= x \bmod a + 0 \\ &= x \bmod a.\end{aligned} \tag{4.2}$$

Recall that given a bucket b, for all records x in that bucket, we know that

$$h(x.\texttt{key}) \bmod B = b$$

as all data in a bucket has the same bucket key. By picking integers $c, k \in \mathbb{N}$ such that $B = ck$, taking modulo c on both sides of the previous equation, and

using Equation (4.2) we have that

$$\begin{aligned} h(x.\texttt{key}) \bmod B &= b \\ (h(x.\texttt{key}) \bmod B) \bmod c &= b \bmod c \\ (h(x.\texttt{key}) \bmod ck) \bmod c &= b \bmod c \\ h(x.\texttt{key}) \bmod c &= b \bmod c. \end{aligned}$$

In other words, we can treat the data as having been bucketed in c buckets, each of which is composed of k of the original B buckets, without actually having to re-bucket the data. For example, the first c-bucket contains records such that

$$h(x.\texttt{key}) \bmod c = 0$$

and is composed as the subset of the B-buckets whose bucket key modulo c is 0.

When at the bucket resolution step at joining time, this property can be used to greatly reduce the number of merge joins computed by picking c such that $B_R = ck_R$ and $B_S = ck_S$ and emitting pairs whose bucket keys modulo c are the same. In other words, c is a common factor of B_R and B_S. The total number of merge joins computed is

$$\frac{B_R B_S}{c}.$$

By picking $c = 1$, this ends up computing the join over all pairs of buckets, whereas we can pick c as the greatest common factor (GCF) of B_R and B_S in order to minimize the number of joins computed: the number of joins computed becomes the minimum common multiple (MCM) of B_R and B_S. Figure 4.4 shows an example. Note that if the two datasets have the same number of buckets, then $c = B_R = B_S$ and there is no replicated data. Conversely, if one dataset only has one bucket, we are computing an equivalent to the map-side join.

With this adjustment, the number of buckets requirement can be removed from the definition of dataset compatibility. In other words, this means that it is possible to calculate B as part of the pipeline. As mentioned in the previous section, an ideal property for buckets is fitting in memory as this enables fast in-memory sorting. Instead of setting the number of buckets, we can hence define a target *bucket size* for each bucket and the amount of buckets required can be computed as the size of the input data divided by the target bucket size. There is, however, a caveat: the amount of replicated data for a given join depends on c, the greatest common factor between the number of buckets

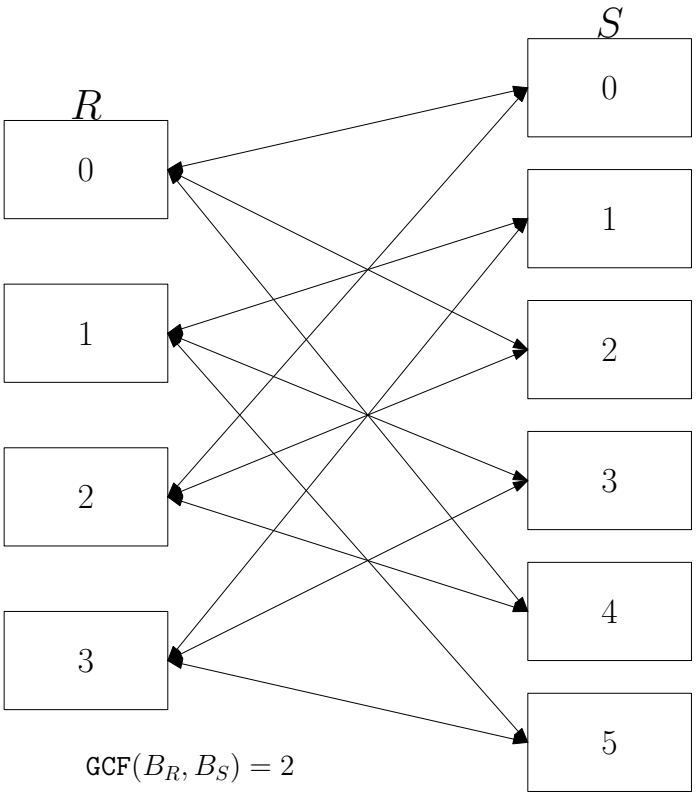


Figure 4.4: Visualization of joining between datasets with different number of buckets.

of two datasets. The lower is c, the higher is the cost of replication. It is hence important to pick numbers of buckets such that c is maximized in order to reduce the amount of replicated. As $c = \texttt{GCF}(B_R, B_S) \leq \texttt{min}(B_R, B_S)$, the replication overhead can be minimized by making sure that the greatest common factor is the minimum between B_R and B_S, which implies that one is a factor of the other. The possible number of buckets can hence be picked as powers of an integer: powers of two are used as that allows finer granularity over other integers. In general, this exponential growth approach is common in computer science and can be seen in several other applications e.g., data structure size (to achieve constant amortized time operations), exponential backoff, and more.

In summary, the *bucket size* adjustment modifies the algorithm as follows:

- The number of buckets parameter has been replaced with a bucket size parameter, which is used to automatically determine the power of two number of buckets that fits the data. The size of input data is computed

as the sum of the serialized size of each record, from which the number of buckets is computed. This happens before the `ExtractBucketKeyFn` operation, which receives the newly computed B as input.

- The `ResolveBucketingFn` `PTransform` at joining time is slightly modified to compute the joins of all buckets whose bucket key modulo c is equal, where c is the minimum between the number of buckets for the two datasets.

The notion of bucket size makes it easier to define skewness. When creating buckets, a bucket is *skewed* if its contents would exceed the allowed bucket size. The second adjustment, *bucket sharding*, splits a bucket into multiple files called *shards*, each of which is locally sorted. The idea behind sharding is the same idea behind partitioning in a hash join: as the data exceeds the target bucket size (i.e., it is too large to fit in memory for a hash join), it is repartitioned. However, there are few limitations of this approach:

1. **Recursive partitioning**: Recursive partitioning as in a normal hash join is not possible, as the `PTransforms` in the Dataflow graph must be fixed at graph construction time.

2. **Data skew**: recursive partitioning does not alleviate data skew. When repartitioning using a different hash function, skewed "hot keys" would have the same hash regardless of the hash function.

For these reasons, records are repartitioned in shards using a round-robin strategy: this allows a bucket to be equally distributed among its component shards. The number of shards for a bucket is hence computed by dividing the total size of records in that bucket over the bucket size. In order to join two buckets, a merge join between all pairs of underlying shards is computed instead. Similarly as the previous step, as each shard is read multiple times, this results in data replication. Note that unlike in the previous scenario, it is not possible to limit the number of joins between shards due to the partitioning strategy used. By collecting other types of data statistics, such as histograms, range partitioning could be used for the shards in order to optimize the shard distribution. However, computing these distributions may be costly and an evaluation of this approach is left as future work.

This adjustment requires modifying both the bucketing and joining phases as follows:

- **Bucketing**: After computing the number of buckets per the *bucket size* adjustment, we compute the total size of records for each bucket. Consequently, we obtain a number of shards for each bucket. In order to not

incur in skewness due to the `GroupByKey` operation, we modify the key K to be a tupled key of $(bucketKey, shardId)$ of which the shard id is assigned in round-robin fashion by each worker. Each composite key corresponds to a shard for a particular bucket, which are constructed to fit in memory of each worker. This allows the `GroupByKey` and sorting to not suffer from skewness. As before, each shard is written out as a single file with all its metadata, including the bucket key for its "parent" bucket and its shard id. As a result, a bucket now might span multiple files.

- **Joining**: The phase is not modified, however the `ResolveBucketingFn` might end up creating more join operations as a result of sharding. Once again, the file patterns for R and S are expanded to match each shard for the two datasets from which the metadata is extracted. Each shard contains its bucket key and is joined with all other shards with a matching bucket key (modulo c if the number of buckets is different).

Listing 5 contains updated pseudocode for the bucketing phase, with the Cloud Dataflow counterpart in Figure 4.5. Some boilerplate code has been removed for brevity: in particular the number of buckets B and shard map M, which contains the number of shards for each bucket, are distributed as a `SideInput` to all workers. The joining pseudocode is unchanged apart from the described changes inside the `ResolveBucketingFn` function.

Listing 5: Bucketing phase for skew-adjusted SMB.

```
1  def BucketingPhase(input, S):
       Data: input is a PCollection⟨V⟩, with V denoting the type of the
             records inside the PCollection. S is the target bucket size.
       Result: PCollection⟨Shard⟨V⟩⟩
2      B ← input
3          .apply(ParDo.of(ComputeSizeFn()))
4          .apply(Sum.globally())
5          .apply(ParDo.of(ComputeNumBucketsFn(S)))
6      M ← input
7          .apply(ParDo.of(ComputeSizeWithBucketKeyFn(B)))
8          .apply(Sum.perKey())
9          .apply(ParDo.of(ComputeNumShardsFn(S)))
10     return input
11         .apply(ParDo.of(ExtractShardedBucketKeyFn(B, M)))
           /* Returns a PCollection⟨K, V⟩, where the key
              K is a composite key consisting of a
              bucket key determined through
              Equation (4.1) over B buckets and a
              shard id assigned in round-robin by
              each worker.  The map M contains the
              number of shards for each possible
              bucket key.                              */
12         .apply(GroupByKey.create())
           /* Returns a PCollection⟨K, Iterable⟨V⟩⟩     */
13         .apply(ParDo.of(SortBucketFns()))
           /* Locally sorts the records in an
              iterable by their join key, creating a
              Shard⟨V⟩ and returning a
              PCollection⟨Shard⟨V⟩⟩.                    */
```

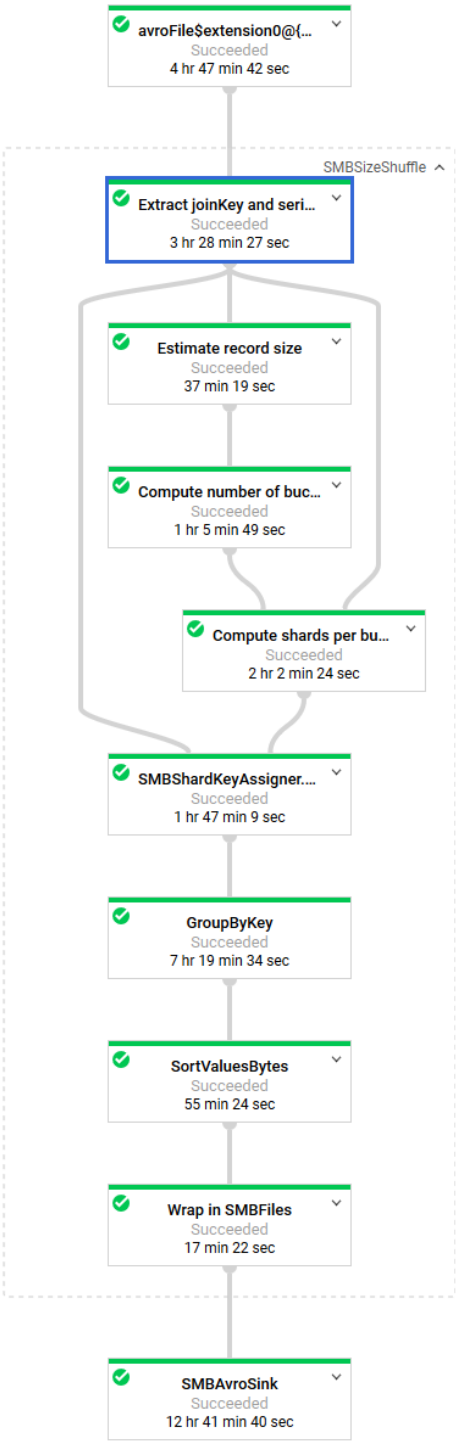


Figure 4.5: Bucketing pipeline for skew-adjusted SMB in Cloud Dataflow.

4.5 Analysis

In analyzing the number of shuffled records of the solutions, consider that Cloud Dataflow does not have utilize *data locality*, unlike Hadoop. Data locality refers to the concept of moving the computation closer to the data, i.e. executing tasks on workers that have data locally, without a network operation. For this reason, reading and writing in Cloud Dataflow consists of a shuffle.

The asymptotic notation for number of shuffled records for SMB is:

1. **Bucketing**: $O(|N|)$, where $|N|$ is the input size. All data is read once $O(|N|)$, shuffled once $O(|N|)$, and rewritten as buckets once $O(|N|)$.

2. **Joining**: $O(|R| + |S|)$, where $|R|$ and $|S|$ are the input sizes of the two relations. The data is only read once $O(|R| + |S|)$, with no shuffle.

The asymptotic number of records shuffled lines up with Hadoop's repartition shuffle, but it suffers from the same problem: skewness and stragglers may delay the pipeline indefinitely.

The asymptotic notation for number of shuffled records for skew-adjusted SMB is:

1. **Bucketing**: $O(|N|)$, where $|N|$ is the input size. All data is read once $O(|N|)$, shuffled once $O(|N|)$, and rewritten as buckets once $O(|N|)$.

2. **Joining**: $O(|R||S|)$, where $|R|$ and $|S|$ are the input sizes of the two relations. The data may be read multiple times to handle different number of buckets and sharded buckets. Consider the worst case scenario in which the two datasets contain records with exactly the same key. All records would end up in one bucket and all its shards would be need to joined pairwise with all the shards from the other bucket. This is the same worst-case asymptotic runtime as hash- or merge- join on a single worker.

However, as explained in Section 2.3, the amount of shuffled data is not the whole story. When stragglers are present, the running time is dominated by the processing speed of the slowest straggler. Table 4.1 compares the amount of shuffled data and running time of the join operations introduced, compared with Hadoop's standard joins, in terms of shuffled data and running time in the best case (no skewness) and worst case (all the data is skewed). For map join, the smaller table R fits in memory: stragglers have no effect as data is not grouped at all. Hence, even if the data were to be completely skewed

with a $O(|R||S|)$ result set, the computation happens across the W workers. Sort merge bucket join mirrors the repartition join: they are the same save for separating the bucketing and joining phases. The worst case running time for skewed datasets is $O(|R||S|)$ as both datasets would need to be joined a single worker. For sort merge bucket join, this means that all data ends up in the same bucket. For skew-adjusted sort merge bucket join, the best case does not include any replication, whether to ensure compatibility or due to sharding. Such a scenario is the same as a sort merge bucket join. When data is highly skewed however, data is replicated up to $O(|R||S|)$ times in order to construct shards that are not skewed, ensuring there are no stragglers. As no stragglers are present, each worker can contribute equally to compute the $O(|R||S|)$ result set like in the map-side join.

Table 4.1: Comparison of network IO and running time between the joins introduced and the joins in Hadoop.

Join Type	**Best Case**		**Worst Case**	
	Network IO	Running Time	Network IO	Running Time
Hadoop Map-side Join	$O(\|R\|W)$	$O\left(\frac{\|R\|\|S\|}{W}\right)$	$O(\|R\|W)$	$O\left(\frac{\|R\|\|S\|}{W}\right)$
Hadoop Repartition Join	$O(\|R\|+\|S\|)$	$O(\|R\|+\|S\|)$	$O(\|R\|+\|S\|)$	$O(\|R\|\|S\|)$
Sort Merge Bucket Join	$O(\|R\|+\|S\|)$	$O(\|R\|+\|S\|)$	$O(\|R\|+\|S\|)$	$O(\|R\|\|S\|)$
Skew-Adjusted Sort Merge Bucket Join	$O(\|R\|+\|S\|)$	$O(\|R\|+\|S\|)$	$O(\|R\|\|S\|)$	$O\left(\frac{\|R\|\|S\|}{W}\right)$

Chapter 5

Results

This section describes evaluation of the SMB solution against standard join techniques. The experiments are run on Cloud Dataflow, using clusters of `n1-standard-4` workers. Each worker has 4 CPUs and 15 GB of available memory. The pipelines are written using the Apache Beam Java SDK, version 2.12.0.

In evaluating the pipelines, three different metrics will be considered: wall clock, CPU hours, and amount of shuffled data. Whereas wall clock provides an intuitive idea of the time spent for a pipeline, it is dependant on the number of workers. Hence, CPU hours are also provided to provide an objective metric of the "cost" of a pipeline. Note they represent the amount of CPU hours allocated to that job, hence the actual number of "active" CPU hours might be less (for example when the CPU is waiting for IO). The last metric consists of the amount of shuffled data in GB as estimated by Cloud Dataflow. Furthermore, recall that Cloud Dataflow does not have data locality: reading/writing of the data is included in the amount of data shuffled. As an additional note, all jobs ran in this section are executed with active autoscaling, but the starting and maximum number of workers is fixed for each experiment. Hence, autoscaling was activated in order to make it easier for Cloud Dataflow to downscale as the pipeline is winding down.

This section evaluates the solution empirically in two scenarios. The first consists of the decryption use case at Spotify. The second uses generated data with increasing degrees of skew.

5.1 Decryption Pipeline

In order to evaluate SMB, we consider the case of decryption of a partition of real events. One of the largest event types processed at Spotify relates to user behavior experiments: it is generated when users interact with Spotify applications. This type of event is selected because it is one of the largest at Spotify. In addition, this event can suffer from skewness when test users are used improperly: a skewed partition can be created if a significant amount of data is generated for a single test user (e.g., when load testing). As an example pipeline that uses this data, analysts could read the events and evaluate engagement with an experiment by user agent, which is an encrypted, sensitive field in the event. In the evaluation scenario we hence use two datasets, which are:

- `Keys`: 1149667449 ($1.15 \cdot 10^9$) records for a total serialized size of 194.87 GB. This dataset corresponds to a subset of all user keys, with each record containing user encryption keys for a specific user.
- `Event`: 6708279884 ($6.7 \cdot 10^9$) events for a total serialized size of 2.86 TB. Corresponds to an hourly partition of encrypted events. All events have a matching user in `Keys`.

The join key for the two datasets is the $user\_id$ field. Pipelines in this section are executed with 128 workers, for a total of 512 CPUs and 1880 GB of memory, with autoscaling enabled. The evaluation scenario consists of a join between `Keys` and `Event` and decryption of a single field, the user agent. The results are listed in Table 5.1.

The first line details the result of the standard join in Dataflow, i.e. a repartition join, between the two datasets. The amount of shuffled data corresponds to just reading the data and performing a `CoGroupByKey` operation.

The SMB join, represented in the second block in Table 5.1, is divided between the bucketing and joining phase. In order to pick the number of buckets, we perform some conservative estimations. Starting from the size of the serialized `Events`, we need to consider that the deserialized records have a JVM-specific overhead, conservatively estimated at $1.4\times$ overhead. In addition, the sorting `PTransform` does not operate in-place, copying the results to a new `Iterable`, another $2\times$ overhead. Landing at total size of 8000 GB for the deserialized data in memory, we decide to create 8096 buckets of approximately one GB each, knowing that each worker CPU has approximately three GB of heap memory after the JVM. For bucketing, the shuffled data includes reading,

Baseline	**Wall Clock**	**CPU Hours**	**Shuffled Data GB**
`Events` $\bowtie$ `Keys`	0:45:53	349.42	6576
SMB Join	**Wall Clock**	**CPU Hours**	**Shuffled Data GB**
Bucketing `Keys`	0:15:00	23.644	845
Bucketing `Events`	0:53:57	423.159	11810
`Events` $\bowtie^{SMB}$ `Keys`	0:19:16	72.767	2974
Skew-Adj. SMB Join	**Wall Clock**	**CPU Hours**	**Shuffled Data GB**
Bucketing `Keys`	0:24:49	35.023	847
Bucketing `Events`	1:11:00	573.293	11810
`Events` $\bowtie^{SMB}_{SkewAdj}$ `Keys`	0:22:44	121.639	5734

Table 5.1: Comparison of joins on the decryption pipeline.

the `GroupByKey` to create buckets, and writing the buckets. The join operation reads each pair of buckets, joins them, and decrypts the user agent field. As expected, the sort merge bucket join is faster at joining time than a regular join as no shuffling happens (apart from reading the data). However, the data has to be preprocessed in the bucketing phase, which is more expensive than joining for all metrics. However, bucketing happens only once for any number of joins. We can compute the number of joins n after which SMB has better performance for a particular metric than regular joins as

$$\texttt{Bucketing} + \texttt{SMBJoin} \cdot n \leq \texttt{Join} \cdot n,$$

which can be solved for n as

$$n \geq \left\lceil \frac{\texttt{Bucketing}}{\texttt{Join} - \texttt{SMBJoin}} \right\rceil. \tag{5.1}$$

By solving Equation (5.1) for CPU hours (including bucketing time for `Event` and `Keys`), we obtain $n = 2$. This means that already after the second sort merge bucket join, the cost of bucketing has already been amortized and the more joins are performed, the fewer CPU hours are used relatively to the baseline. The same applies for the amount of shuffled data, which becomes less after $n = 4$ joins. A visualization of the total CPU time after n joins for the three different strategies is available in Figure 5.1.

As a side-effect of bucketing, the Avro format can better leverage data compression: the bucketed events have a size of 1088 GB after compression,

compared to the original non-bucketed data with 1506 GB after compression. The opposite is true for keys: as the data was originally stored in fewer files and each `user_id` in `Keys` is unique, rearranging it into the buckets slightly increased the size after compression, from 120.1 to 122.5 GB.

The third block describes results for skew-adjusted SMB join. Instead of picking the number of buckets, we simply set a bucket size of 300 MB in order for each worker to have ample memory to sort it. The bucket size will be used by the pipeline to automatically determine the number of buckets (and shards) to use. The `Keys` are bucketed in 1024 buckets, each with one shard, whereas `Events` are bucketed in 16384 buckets, one of which has 2 shards for a total of 16385 files. As the data fits evenly in the buckets (except for one), this dataset has very low skew. At joining time, each shard from `Keys` is replicated 16 times due to the different number of buckets. One shard is replicated one additional time in order to handle the slightly skewed bucket. Solving Equation (5.1) for CPU hours, we obtain $n = 3$. Respectively for amount of shuffled data, $n = 16$. As the dataset has very low skew, skew-adjusted SMB performs worse than regular sort merge bucket join due to the price of handling skew (even if it is not present).

As an additional form of comparison, we can compare skew-adjusted SMB in terms of its overhead against regular SMB (visualized in Figure 5.2). For CPU hours, the overhead starts with approximately 40% and grows to 67% as the number of joins increases asymptotically. In terms of shuffled data, the overhead starts with 19% and increases to 93% as the number increases asymptotically. In order to more thoroughly evalaute skew-adjusted SMB, the next section repeats the analysis for generated datasets with different degrees of skew.

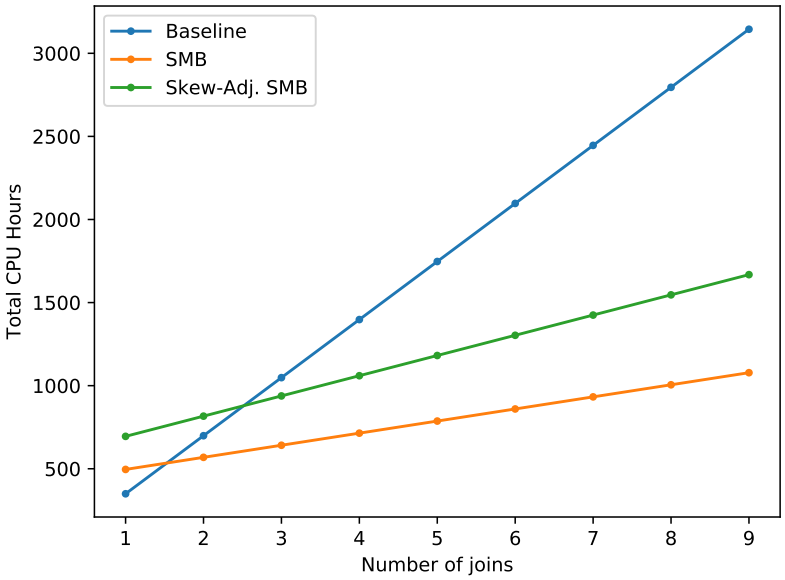


Figure 5.1: Total CPU hours after n joins for regular join, SMB join, and skew-adjusted SMB join.

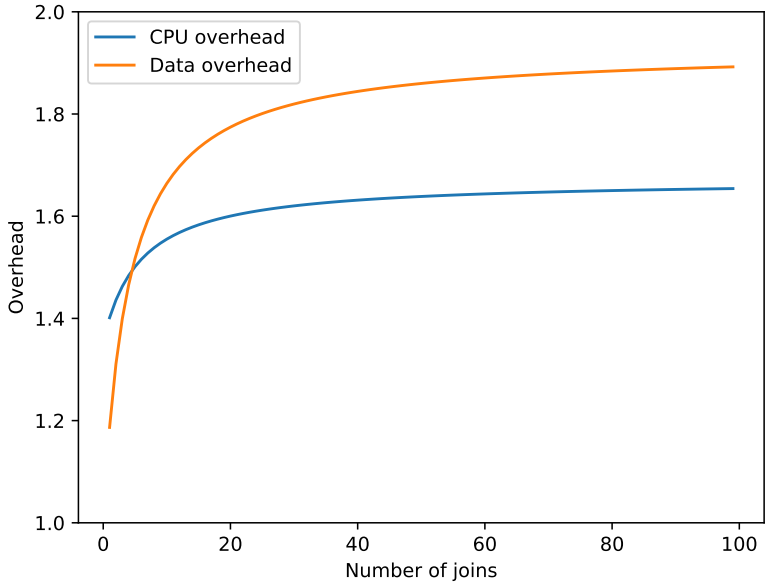


Figure 5.2: Overhead of skew-adjusted SMB against regular SMB for CPU and shuffled data over number of joins.

5.2 Generated Data

The power law family of distributions is commonly used to model frequencies. In this project, we generate data whose keys follow a Zipf's law. This law states that the frequency of a key is inversely proportional to its rank, i.e. the i-th key has frequency that is

$$\frac{1}{i^s}$$

times the most frequent key, for some shape parameter s. The higher is s, the more skewed are the frequencies. To simulate the decryption pipeline, we generate two types of datasets, respectively `Keys` and `Events`. The first dataset, `Keys`, consists of 10^9 (one billion) records containing two fields: an unique integer join key called id (4 bytes) and a random 36 byte string simulating the decryption key for that user. The `Events` dataset consists of $6 \cdot 10^9$ (six billion) records also containing two fields: the id of a user and an event payload consisting of 96 bytes. In an event partition, the unique keys for a single hour is in the order of tens of millions, hence the number of unique id in `Event` has been approximated with $50 \cdot 10^6$ unique keys. This means that not all keys in `Keys` have a correspondence with the `Event` dataset. In order to test the effect of skewness, we generate the `Event` dataset 15 times, for linearly spaced values of $s \in [0.0, 1.4]$. Figure 5.3 plots the frequency of the first million most frequent keys for a subset of the values s. When $s = 0$ all keys have an equal frequency of 120. For $s = 0.2$, the first five keys have frequencies $[3327, 2896, 2670, 2521, 2411]$. As s increases, the first key has a frequency of 86482 for $s = 0.4$, 1999427 (approximately two millions) for $s = 0.6$, 35534777 (approximately 36 millions) for $s = 0.8$ up to 1933322357 (approximately two billions, roughly one third of the dataset) for $s = 1.4$.

We evaluate the sort merge bucket join and skew-adjusted sort merge bucket join against a regular join in Cloud Dataflow, which is equivalent to an Hadoop's repartition join. Pipelines in this section are executed with 32 workers, for a total of 128 CPUs and 480 GB of memory, with autoscaling enabled. The total serialized size for the `Events` datasets is approximately 640 GB, while the total serialized size for the `Keys` dataset is approximately 50 GB.

5.2.1 Baseline

The baseline consists of a simple pipeline which reads both datasets and computes a join between them, summarized in Table 5.2. The join performance appears to be approximately constant until $s > 0.7$. As the data becomes

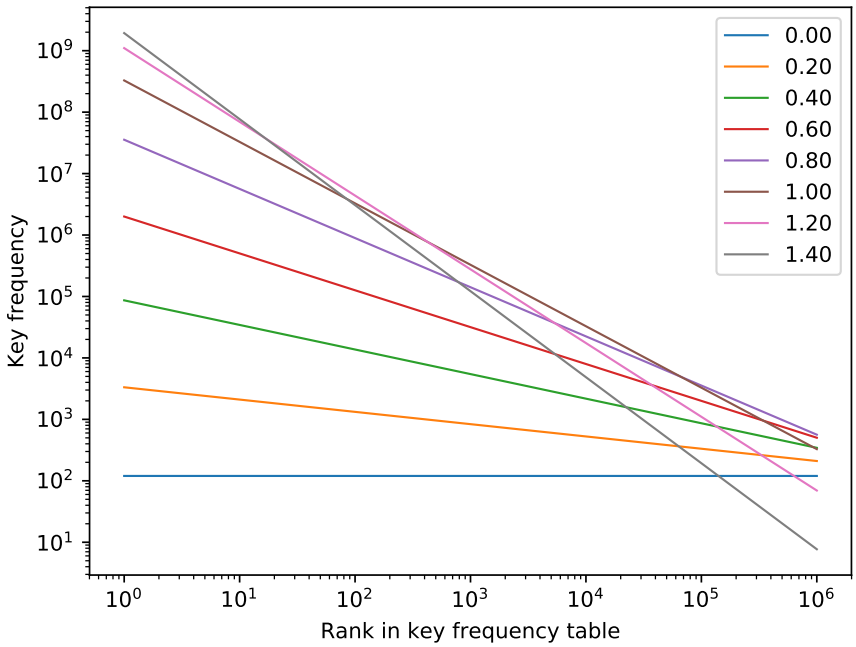


Figure 5.3: Frequency distribution for the first million most frequent keys in generated `Event` for different values of s.

more and more skewed, we observe a steep increase in the time spent for the job. By inspecting the Dataflow graph, as seen in Figure 5.4, we can confirm our suspicions that a straggler is present. Out of the 10^9 unique join keys, a worker is stuck in the `GroupByKey` (GBK, highlighted in blue) grouping all the data for the last skewed key, slowing down pipeline progress. The remainder 999999999 keys have already been processed. As expected from Section 4.5, the amount of shuffled data remains approximately constant.

Table 5.2: Join between `Keys` and `Events` for different values of s.

`Events` ⋈ `Keys` **Repartition Join** for diff. values of s	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read + GBK)
0.0	0 : 27 : 12	52.343	1485.22
0.1	0 : 27 : 27	53.175	1484.15
0.2	0 : 28 : 30	55.356	1483.98
0.3	0 : 27 : 33	52.790	1483.48
0.4	0 : 27 : 24	52.739	1482.78
0.5	0 : 27 : 56	53.845	1481.83
0.6	0 : 27 : 26	53.006	1480.49
0.7	0 : 28 : 12	54.059	1478.47
0.8	0 : 32 : 16	63.061	1475.36
0.9	0 : 44 : 19	90.093	1465.99
1.0	1 : 18 : 00	161.499	1459.73
1.1	2 : 09 : 00	267.030	1442.35
1.2	3 : 44 : 00	469.501	1436.95
1.3	4 : 47 : 00	602.886	1435.39
1.4	5 : 58 : 00	750.614	1429.68

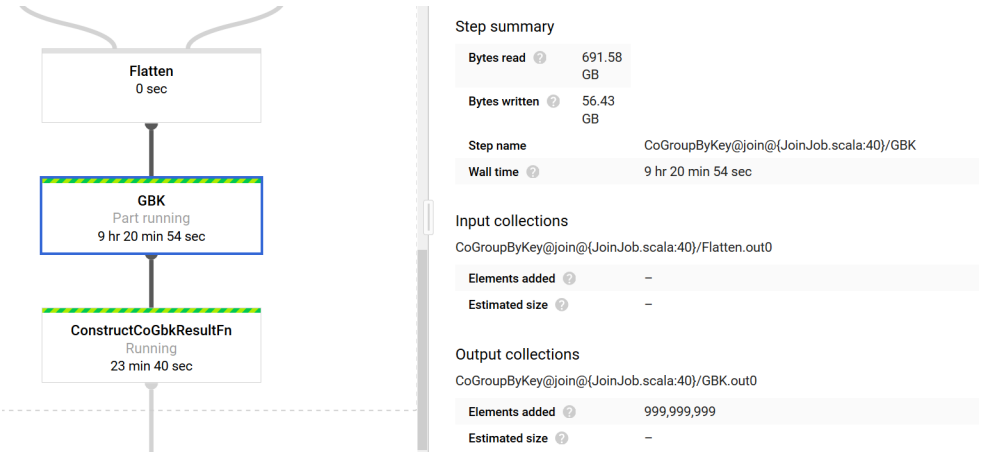


Figure 5.4: An example of a straggling join.

5.2.2 SMB Join

Wall clock, CPU hours, and size of shuffled data for the bucketing phase for `Keys` and `Events` are available in Table 5.3 and in Table 5.4 respectively for the joining phase. The pipelines for $s = 0.9$ and $s = 1.0$ fail in the bucketing step due to lack of progress when sorting. Due to the degree of skew, the workers responsible for sorting the skewed buckets were straggling and were not outputting any records for over 30 minutes, causing a failure in Cloud Dataflow. Figure 5.5 plots the total CPU hours of a single regular join (in Table 5.2) and a single SMB join. The collected metrics can be compared to the baseline using Equation (5.1). We obtain that SMB is more advantageous than regular joins in terms of CPU hours starting with the second SMB join ($n = 2$), for all values of s. Note that in this scenario we do not include the bucketing step for `Keys`, as that happens only once for all 15 values of s. It can also happen in parallel as bucketing of the `Events`, in addition to requiring less than one tenth of the time. Repeating the same analysis for the amount of shuffled data, we obtain that SMB join shuffles fewer data after the fourth join ($n = 4$) for all s.

In summary, this means that sort merge bucket join has better performance than regular join after performing two joins and shuffles fewer data after four joins. However, this join is not robust to higher degrees of skewness as some tested scenarios fail.

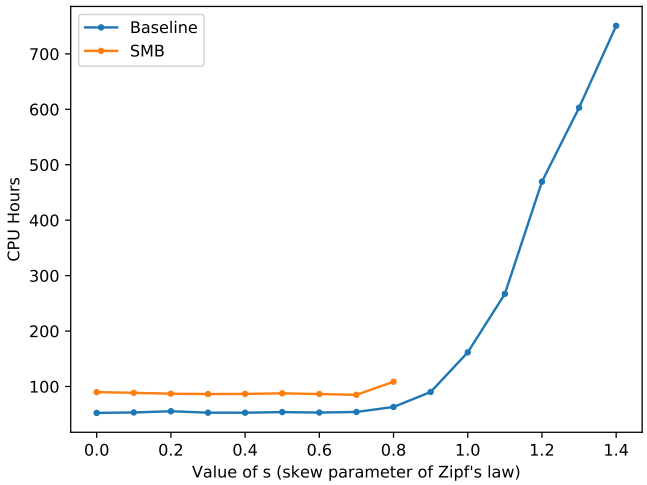


Figure 5.5: Comparison of CPU hours of SMB (bucketing and joining) against one regular join for different values of s (one single join operation).

Table 5.3: Bucketing phase of sort merge bucket join for `Keys` and `Events`.

Bucketing `Keys`	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read + GBK + Write)
n/a	0 : 16 : 13	7.271	205.33
Bucketing `Events` for diff. values of s	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read + GBK + Write)
0.0	0 : 40 : 04	77.496	2595.42
0.1	0 : 39 : 25	76.100	2593.58
0.2	0 : 38 : 42	74.846	2593.50
0.3	0 : 38 : 56	74.683	2592.87
0.4	0 : 38 : 53	75.697	2591.87
0.5	0 : 39 : 03	76.454	2590.48
0.6	0 : 38 : 21	74.901	2588.48
0.7	0 : 38 : 08	73.871	2585.38
0.8	0 : 48 : 50	96.901	2580.68
0.9	*failed*	*failed*	*failed*
1.0	*failed*	*failed*	*failed*

Table 5.4: Joining phase of sort merge bucket join for `Keys` and `Events`.

`Events` ⋈ `Keys` **SMB Join** for diff. values of s	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read)
0.0	0 : 08 : 41	12.368	620.31
0.1	0 : 09 : 07	12.482	619.84
0.2	0 : 08 : 50	12.148	619.75
0.3	0 : 08 : 34	11.692	619.52
0.4	0 : 08 : 08	10.944	619.18
0.5	0 : 08 : 12	11.204	618.71
0.6	0 : 08 : 00	11.531	618.05
0.7	0 : 08 : 05	11.056	617.01
0.8	0 : 08 : 28	11.711	615.46
0.9	*n/a*	*n/a*	*n/a*
1.0	*n/a*	*n/a*	*n/a*

5.2.3 Skew-Adjusted SMB Join

Wall clock, CPU hours, and size of shuffled data for the skew-adjusted SMB join for `Keys` and `Events` are available in Table 5.5 for bucketing (note the additional column, representing the total number of shards written out) and in Table 5.6 for joining, with the last column representing the value of n in Equation (5.1) for CPU hours and shuffled data respectively.

When compared with regular joins for $s < 0.8$, skew-adjusted SMB has better performance in terms of CPU hours after three joins and shuffles less data after five. As skewness quickly increases, skew-adjusted SMB uses fewer CPU hours than a single regular join for $s \geq 1.0$, as can be seen in Figure 5.6. This advantage in processing time has a tradeoff in amounts of data shuffled: as data becomes more skewed, skew-adjusted SMB replicates more and more shards. For the highest degree of skew tested, it still shuffles fewer data after 10 joins.

When compared with regular sort merge bucket join, for $s < 0.8$, the bucketing operation has a overhead of approximately 25%. For $s = 0.8$, the bucketing takes the same amount of CPU hours. For $s > 0.8$, unlike SMB join, skew-adjusted SMB handles all degrees of skewness tested. Skew-adjusted SMB join also relaxes the notion of compatibility to allow joining with datasets with different numbers of buckets.

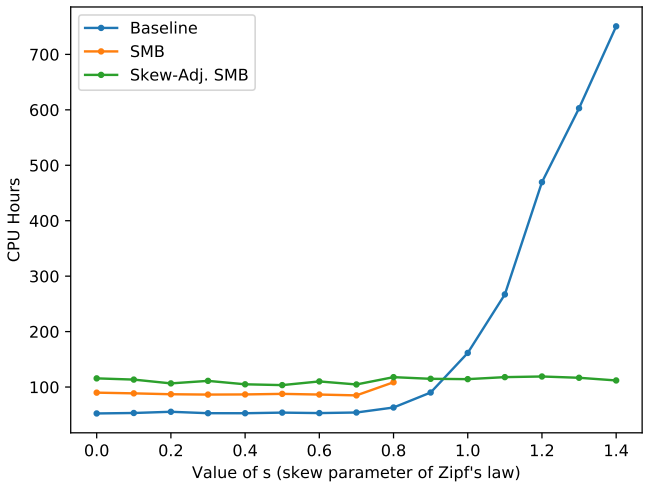


Figure 5.6: Like Figure 5.5, but with skew-adjusted SMB in green (one single join operation).

Table 5.5: Bucketing phase of skew-adjusted SMB join for `Keys` and `Events`.

Bucketing `Keys`	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read + GBK + Write)	**Shards**
n/a	0 : 23 : 59	11.275	206.83	512
Bucketing `Events` for diff. values of s	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read + GBK + Write)	**Shards**
0.0	0 : 48 : 22	97.423	2606.94	4096
0.1	0 : 47 : 07	95.113	2605.09	4096
0.2	0 : 44 : 05	88.211	2605.04	4096
0.3	0 : 46 : 27	93.051	2604.38	4097
0.4	0 : 43 : 03	86.183	2603.39	4105
0.5	0 : 42 : 28	85.178	2602.03	4200
0.6	0 : 45 : 21	90.889	2600.34	4484
0.7	0 : 42 : 37	85.222	2597.00	4774
0.8	0 : 48 : 17	97.524	2591.98	4927
0.9	0 : 47 : 02	93.961	2581.75	5227
1.0	0 : 46 : 00	92.573	2572.81	5691
1.1	0 : 48 : 37	95.683	2563.15	6359
1.2	0 : 49 : 11	96.485	2539.17	6896
1.3	0 : 48 : 16	93.908	2538.45	7293
1.4	0 : 46 : 47	89.342	2540.40	7528

Table 5.6: Joining phase of skew-adjusted SMB join for `Keys` and `Events`.

`Events` ⋈ `Keys` **Skew-Adj. SMB Join** for diff. values of s	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Replicated Read)	n (CPU – IO)
0.0	0 : 12 : 14	18.194	899.75	3 - 5
0.1	0 : 12 : 04	18.299	899.28	3 - 5
0.2	0 : 12 : 11	18.354	899.19	3 - 5
0.3	0 : 12 : 24	18.062	899.04	3 - 5
0.4	0 : 12 : 21	18.741	899.32	3 - 5
0.5	0 : 12 : 30	18.282	906.26	3 - 5
0.6	0 : 13 : 05	19.209	927.74	3 - 5
0.7	0 : 12 : 48	19.409	949.31	3 - 5
0.8	0 : 13 : 27	20.293	959.69	3 - 6
0.9	0 : 13 : 34	20.866	981.48	2 - 6
1.0	0 : 13 : 49	21.654	1014.86	1 - 6
1.1	0 : 18 : 28	22.171	1064.82	1 - 7
1.2	0 : 17 : 36	22.527	1098.79	1 - 8
1.3	0 : 17 : 26	22.777	1128.53	1 - 9
1.4	0 : 17 : 48	22.522	1152.54	1 - 10

An additional comparison can be seen in Figure 5.7. This graph plots the value of n, the number of joins after which skew-adjusted SMB is better than regular joins in terms of CPU or shuffled data, for all values of s. As mentioned previously, after $s \geq 1.0$, skew-adjusted SMB is better than a single regular join in terms of CPU hours. This performance is achieved by replicating data: in other words, there is a tradeoff between CPU hours and amount of data shuffled. As skewness increases, the n for shuffled data increases accordingly as more data is being replicated.

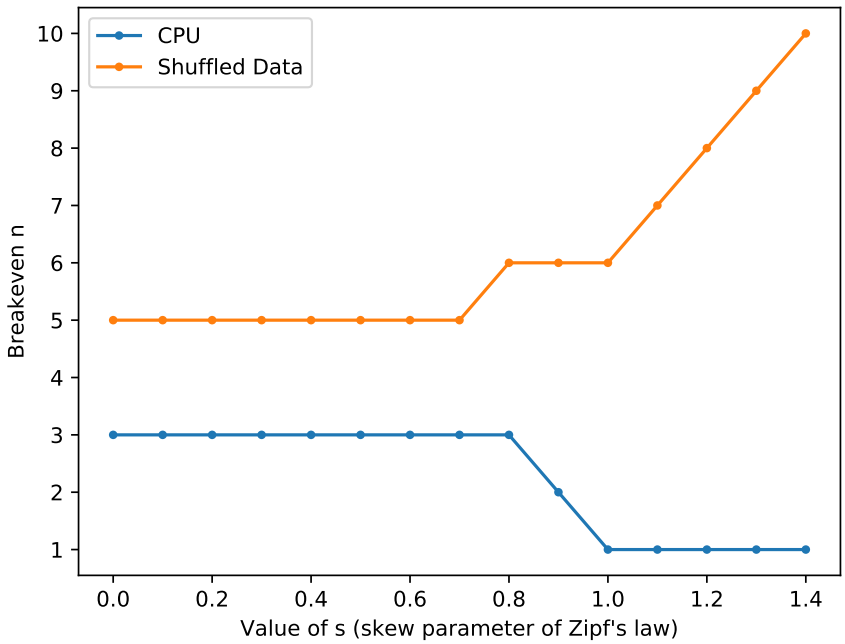


Figure 5.7: Value of breakeven n for skew-adjusted SMB compared against the baseline, for different values of s.

Chapter 6

Conclusions

This work uses sort merge buckets to optimize repeated joins at massive scale. Repeated joins may arise e.g., in data decryption. The sort merge buckets solution has been extended to skew-adjusted sort merge buckets, which is robust to different input sizes, varying degrees of skew, and is more flexible than regular SMB join.

Both SMB and skew-adjusted SMB join consist of a preprocessing step which reorganizes data and joining step which exploits the structure of the data. The preprocessing step is backwards-compatible: a user that is not aware of the structure can still perform a regular join operation. When evaluated on the decryption pipeline at Spotify, sort merge buckets achieves lower compute costs after the second join and a smaller amount of shuffled data after the fourth.

The skew-adjusted SMB join introduces some overhead in order to automatically determine optimal sharding strategies and mitigate the effects of skew. It does so by replicating shards in order to have a broader versatility in joining with datasets bucketed with a different number of buckets and in order to tackle skewed buckets. When data is not skewed, skew-adjusted SMB has an overhead when compared to regular sort merge buckets: however, it is still more compute-efficient than regular joins after the third join operation. As an additional advantage, skew-adjusted SMB is robust to fluctuating data size in addition to skew and has a lower barrier of entry: the programmer does not need to estimate properties of the data or understand how the number of buckets ties into the inner workings of the procedure.

Results from previous chapter show how we can combine the two techniques to optimize different join scenarios based on different degrees of skew:

- **Low Skew** ($0 \leq s < 0.8$): If joining more than two times, sort merge buckets join has better performance than a regular join.

- **Medium Skew** ($0.8 \leq s < 1.0$): If joining more than three times, skew-adjusted SMB has better performance than a regular join.
- **High Skew** ($s \geq 1.0$): Regardless of the number of joins, skew-adjusted SMB has a better performance.

The experiments have been conducted on Cloud Dataflow through Apache Beam, the reference implementation of the Dataflow model. However, no runner-specific detail are used, making this solution broadly applicable to all of Beam's runners. As the solution simply consists of pre-partitioning and reordering data, it is expected that it is applicable to many other MapReduce-inspired platforms.

As a consequence of the promising results for Spotify's decryption use case as evidenced in Section 5.1, a patch introducing sort merge bucket join has been proposed to Apache Beam[1], with the help of more experienced developers at Spotify. Currently, this patch only contains the initial sort merge bucket solution.

The code used to run the experiments is available publicly at `https://github.com/Illedran/beam-smbjoin`, including the code for generating different types of skewed data in Beam. The sort merge bucket implementation is written in Apache Beam, while the pipelines are written as thin Scala wrapper over the implemented `PTransforms` using Scio.

6.1 Limitations

SMB presents some limitations. First of all, this type of optimization as described in this work is limited to joins with equality condition. This is a consequence of using a hash function for partitioning: "theta" joins, which use different conditions for joining records, are not supported, as records that would join might end up in different buckets.

When it comes to the join type, all types (inner, left/right, outer) are supported if the number of buckets is the same. If the number of buckets is different, or buckets are skewed, non-inner joins will produce more records due to data duplication, with incorrect results.

While the target bucket size is a more understandable parameter compared to the number of buckets, picking "good" values might still prove to be tricky. As a reminder, the target bucket size should be small enough that each bucket fits in memory. However, picking a too small bucket size might increase the time

[1] `https://issues.apache.org/jira/browse/BEAM-6766`

spent when resolving the bucketing metadata and create additional overhead in opening and reading a large number of files as opposed to reading a smaller number of files which are larger. If possible, the target bucket size should be picked with awareness of the block size of the underlying distributed filesystem in order to minimize internal fragmentation.

6.2 Future Work

The work presents several different areas of possible improvement.

In this work, the bucketing metadata is stored in each file. A unified metadata store or format can broadly increase compatibility between different data formats and encourage widespread adoption. For example, the metadata could be stored in a single file together with the data, e.g., a JSON file. The file can store the metadata for each file in its directory, simplifying matching and the bucketing resolution step. As a result, the bucketing resolution step could simply become reading the two metadata files without inspecting each single file. Alternatively, the bucketing metadata can be stored separately in a metadata store. For example, this could enable even easier access to the optimization as compatibility information is handled by the metadata store.

As the data in each bucket is sorted, future work could explore the feasibility of creating buckets incrementally. For example, this could be used to create daily aggregates from hourly partitions. Moreover, incremental buckets could be even more suitable for use cases such as streaming. The advantages of data being sorted can be extended to many different operations which could also exploit this structure such as look up operations or data deduplication.

As described in Section 4.4, records in a bucket are partitioned to its shards using round-robin, which results in shards being replicated depending on the degree of skewness. Different types of partitioning, such as range partitioning, could be used to reduce the degree of data replication by e.g., replicating only the shards that are responsible for a particular key. However, the advantages of more advanced partitioning strategies must be weighted against their cost: for example computing histograms required for range partitioning might further increase the overhead required to handle skewness. In general, the problem can be formulated as a trade-off between the computational and shuffling cost, with more advanced partitioning strategies requiring larger computational costs in preprocessing.

Bibliography

[1] Spotify AB. *Spotify – Company Info*. Dec. 31, 2018. URL: https://newsroom.spotify.com/company-info/ (visited on 02/26/2019).

[2] Tyler Akidau et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: *Proceedings of the VLDB Endowment* 8.12 (Aug. 1, 2015), pp. 1792–1803. ISSN: 21508097. DOI: 10.14778/2824032.2824076. URL: http://dl.acm.org/citation.cfm?doid=2824032.2824076 (visited on 04/09/2019).

[3] Foto N. Afrati and Jeffrey D. Ullman. "Optimizing joins in a map-reduce environment". en. In: *Proceedings of the 13th International Conference on Extending Database Technology - EDBT '10*. Lausanne, Switzerland: ACM Press, 2010, p. 99. ISBN: 978-1-60558-945-9. DOI: 10.1145/1739041.1739056. URL: http://portal.acm.org/citation.cfm?doid=1739041.1739056 (visited on 02/06/2019).

[4] YongChul Kwon et al. "A study of skew in mapreduce applications". In: *Open Cirrus Summit* 11 (2011).

[5] Bram Leenders. *Scalable User Privacy*. Spotify Labs. Sept. 18, 2018. URL: https://labs.spotify.com/2018/09/18/scalable-user-privacy/ (visited on 06/13/2019).

[6] *Why Energy Is A Big And Rapidly Growing Problem For Data Centers*. URL: https://www.forbes.com/sites/forbestechcouncil/2017/12/15/why-energy-is-a-big-and-rapidly-growing-problem-for-data-centers/#433379815a30.

[7] International Data Corporation. *The Digitization of the World - From Edge to Core*. URL: https://www.seagate.com/gb/en/our-story/data-age-2025/ (visited on 05/24/2019).

[8] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *OSDI’04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System”. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, 2003, pp. 20–43.

[10] Konstantin Shvachko et al. “The Hadoop Distributed File System”. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). Incline Village, NV, USA: IEEE, May 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972. URL: http://ieeexplore.ieee.org/document/5496972/ (visited on 05/24/2019).

[11] Craig Chambers et al. “FlumeJava: easy, efficient data-parallel pipelines”. In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI ’10*. the 2010 ACM SIGPLAN conference. Toronto, Ontario, Canada: ACM Press, 2010, p. 363. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806638. URL: http://portal.acm.org/citation.cfm?doid=1806596.1806638 (visited on 04/09/2019).

[12] *Apache Beam*. URL: https://beam.apache.org/ (visited on 02/26/2019).

[13] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing”. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI’12. San Jose, CA: USENIX Association, 2012, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=2228298.2228301.

[14] Tyler Akidau et al. “MillWheel: fault-tolerant stream processing at internet scale”. In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 27, 2013), pp. 1033–1044. ISSN: 21508097. DOI: 10.14778/2536222.2536229. URL: http://dl.acm.org/citation.cfm?doid=2536222.2536229 (visited on 04/09/2019).

[15] Matei Zaharia et al. “Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters”. In: *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’12. Boston, MA: USENIX Association, 2012, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=2342763.2342773.

[16] G. Graefe. “Sort-merge-join: an idea whose time has(h) passed?” In: *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*. 1994 IEEE 10th International Conference on Data Engineering. Houston, TX, USA: IEEE, 1994, pp. 406–417. ISBN: 978-0-8186-5402-2. DOI: 10.1109/ICDE.1994.283062. URL: http://ieeexplore.ieee.org/document/283062/ (visited on 02/06/2019).

[17] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. 5th ed. New York, NY, USA: McGraw-Hill, Inc., 2006. ISBN: 9780072958867.

[18] Surajit Chaudhuri. “An Overview of Query Optimization in Relational Systems”. In: *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS ’98. Seattle, Washington, USA: ACM, 1998, pp. 34–43. ISBN: 0-89791-996-3. DOI: 10.1145/275487.275492. URL: http://doi.acm.org/10.1145/275487.275492.

[19] Alok Aggarwal and Jeffrey Vitter S. “The input/output complexity of sorting and related problems”. In: *Communications of the ACM* 31.9 (Aug. 1, 1988), pp. 1116–1127. ISSN: 00010782. DOI: 10.1145/48529.48535. URL: http://portal.acm.org/citation.cfm?doid=48529.48535 (visited on 05/26/2019).

[20] G. Graefe, A. Linville, and L.D. Shapiro. “Sort vs. hash revisited”. In: *IEEE Transactions on Knowledge and Data Engineering* 6.6 (Dec. 1994), pp. 934–944. ISSN: 10414347. DOI: 10.1109/69.334883. URL: http://ieeexplore.ieee.org/document/334883/ (visited on 04/08/2019).

[21] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. “Massively parallel sort-merge joins in main memory multi-core database systems”. In: *Proceedings of the VLDB Endowment* 5.10 (June 1, 2012), pp. 1064–1075. ISSN: 21508097. DOI: 10.14778/2336664.2336678.

URL: http://dl.acm.org/citation.cfm?doid=2336664.2336678 (visited on 02/06/2019).

[22] Changkyu Kim et al. "Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs". In: *Proceedings of the VLDB Endowment* 2.2 (Aug. 1, 2009), pp. 1378–1389. ISSN: 21508097. DOI: 10.14778/1687553.1687564. URL: http://dl.acm.org/citation.cfm?doid=1687553.1687564 (visited on 02/06/2019).

[23] Spyros Blanas et al. "A Comparison of Join Algorithms for Log Processing in MaPreduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 975–986. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807273. URL: http://doi.acm.org/10.1145/1807167.1807273.

[24] Jimmy Lin et al. "The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce". In: *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*. Vol. 1. ACM Boston, MA, USA. 2009, pp. 57–62.

[25] Benjamin Gufler et al. "Handling Data Skew in MapReduce." In: *Closer* 11 (2011), pp. 574–583.

[26] YongChul Kwon et al. "SkewTune: mitigating skew in mapreduce applications". en. In: *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12*. Scottsdale, Arizona, USA: ACM Press, 2012, p. 25. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213840. URL: http://dl.acm.org/citation.cfm?doid=2213836.2213840.

[27] M. Al Hajj Hassan, M. Bamha, and F. Loulergue. "Handling Data-skew Effects in Join Operations Using MapReduce". In: *Procedia Computer Science* 29 (2014), pp. 145–158. ISSN: 18770509. DOI: 10.1016/j.procs.2014.05.014. URL: https://linkinghub.elsevier.com/retrieve/pii/S1877050914001914 (visited on 05/27/2019).

[28] Ganesh Ananthanarayanan et al. "Effective Straggler Mitigation: Attack of the Clones". In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 185–198. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ananthanarayanan.

[29] Igor Maravić. *Spotify's Event Delivery – The Road to the Cloud (Part I)*. Spotify Labs. Feb. 25, 2016. URL: https://labs.spotify.com/2016/02/25/spotifys-event-delivery-the-road-to-the-cloud-part-i/ (visited on 06/13/2019).

[30] *Scio*. URL: https://github.com/spotify/scio/ (visited on 02/26/2019).

[31] Austin Appleby. *MurmurHash*. Mar. 1, 2011. URL: https://sites.google.com/site/murmurhash/ (visited on 06/20/2019).